Almost every slightly complex piece of software will eventually require the use of threads to achieve the required goals. It could be due to performance, collaboration, polling, long execution tasks or other programming reasons. In the specific case of components, every call will run in a different thread, so the program is, by design using threads, however, there are still reasons to create new threads, for instance to schedule and execute tasks at specific times.

In principle, for clients and components, the developer is free to use the threading mechanism they prefer. In particular, for portability, ACS provides managed threading facilities to control C++ threads lifecycle, which could be used by clients and components. A second benefit of using managed threads in containers, is that the created threads will get destroyed along with the termination of the component or the container itself, ensuring proper release of resources if, for any reason, they were not manually released before.

The ACS Threads were created before C++ supported threads, hence it is based in ACE/TAO's spawn function, which, for Linux (and other OSs), is based on pthread. Considering that C++ is now offering a proper threading standard it makes sense to move forward in this direction. This would also allow to provide a leaner interface to make developers life easier.

- Scope
    - Threading in the different programming languages
    - ACS Managed Threading in C++
- Duration: 15 minutes
- ACS Threads.pdf

Thread Class Definition

acsThreadTest.h

```
#ifndef _ACS_THREAD_TEST_H
#define _ACS_THREAD_TEST_H

#include "acsThread.h"

class TestACSThread : public ACS::Thread
{
  public:
    TestACSThread(const ACE_CString& name,
        const ACS::TimeInterval& responseTime=ThreadBase::defaultResponseTime,
                const ACS::TimeInterval& sleepTime=ThreadBase::defaultSleepTime, bool del=false
    ) : ACS::Thread(name, responseTime, sleepTime, del) {
            ACS_TRACE("TestACSThread::TestACSThread");
            loopCounter_m = 0;
        }

    TestACSThread(const ACE_CString& name, const ACS::TimeInterval& responseTime,
        const ACS::TimeInterval& sleepTime, bool del, const long _thrFlags
        ) : ACS::Thread(name, responseTime, sleepTime, del, _thrFlags) {
            ACS_TRACE("TestACSThread::TestACSThread");
            loopCounter_m = 0;
        }

    ~TestACSThread() {
            ACS_TRACE("TestACSThread::~TestACSThread");
            terminate();
        }

    virtual void runLoop() {
            if (loopCounter_m==100) {
            exit();
            }
        ACS_LOG(LM_SOURCE_INFO, "TestACSThread::runLoop", (LM_INFO, "%s: runLoop (%d)", getName().c_str(),
loopCounter_m));
        ++loopCounter_m;
        }

  protected:
    int loopCounter_m;
};

#endif /* end _ACS_THREAD_TEST_H */
```

Client Code Using Thread Manager

<clientFile>.cpp

```
#include "acsThreadManager.h"
#include "acsThreadTest.h"


int main(int argc, char *argv[]) {
    LoggingProxy logger_m(0, 0, 31);
    LoggingProxy::init(&logger_m);
    ACS_CHECK_LOGGER;


    //Obtain Thread Manager
    ACS::ThreadManager tm(getNamedLogger("ThrMgrLogger"));


    //Create thread
    TestACSThread* test = tm.create<TestACSThread>("TestThread");


    //Resume execution
    test.resume();


    //Wait reasonable time...
    sleep(10);


    //Release Thread Resources
    tm.destroy(test);


    return 0;
}
```

Component Code Using Thread Manager

<idl>.idl

```
#ifndef _<idl>_IDL_
#define _<idl>_IDL_

#include <acscomponent.idl>

module <module> {
    interface <interface> : ACS::ACSComponent {
        void resume();
        void pause();
    };
};
#endif
```

<classFile>.h

```
#ifndef _<classFile>_H
#define _<classFile>_H

#include <acscomponentImpl.h>
#include <<idl>S.h>

#include "acsThreadTest.h"

class <class> : public virtual acscomponent::ACSComponentImpl, public virtual POA_<module>::<interface>
{
  public:
    <class>(const ACE_CString& name, maci::ContainerServices* containerServices);
    virtual ~<class>();
    void execute();
    void resume();
    void pause();
    void cleanUp
  protected:
    TestACSThread* test = NULL;
};
#endif
```

<classFile>.cpp

```
//Component code...
#include <<classFile>.h>

<class>::<class>(const ACE_CString& name, maci::ContainerServices* containerServices) : ACSComponentImpl(name,
containerServices) {
    ACS_TRACE("<class>::<class>");
}

<class>::~<class>() {
    ACS_TRACE("<class>::~<class>");
}

void <class>::execute() {
    test = getContainerServices()->getThreadManager()->create<TestACSThread>("ThreadTest");
}

void <class>::resume() {
    test.resume();
}

void <class>::pause() {
    test.suspend();
}

void <class>::cleanUp() {
    getContainerServices()->getThreadManager()->destroy(test);
}

/* --------------- [ MACI DLL support functions ] -----------------*/
#include <maciACSComponentDefines.h>
MACI_DLL_SUPPORT_FUNCTIONS(<class>)
/* ----------------------------------------------------------------*/
```

- Dependency with ACE/TAO spawn / pthread
- Upgrading C++ Threading Technology (std::thread)
- Managed Threading for other languages

- Initial motivations to implement ACS Threads in C++
  - Portability, abstraction for complex interfaces, configuration capabilities
- While them may no longer be concerns:
  - ✅ Portability: std::thread, boost::thread

    - ✅ Abstraction for complex interfaces: std::thread, boost::thread
    - ✅ Configuration capabilities: boost::thread
- There was one concern that is still valid and that ACS Threads take care of naturally:
    - Setting up the logger for each created thread
- There was an advantage that the current implementation has:
    - Thread Manager resources deallocation on component / container exit
- One alternative would be to design a registration for lifecycle management
    - In C++ it should configure the loggers for the thread
    - This could also be desirable for other language implementations