C++ Template-Techniken für verteilte Softwaresysteme

Dr.-Ing. Rainer Niekamp

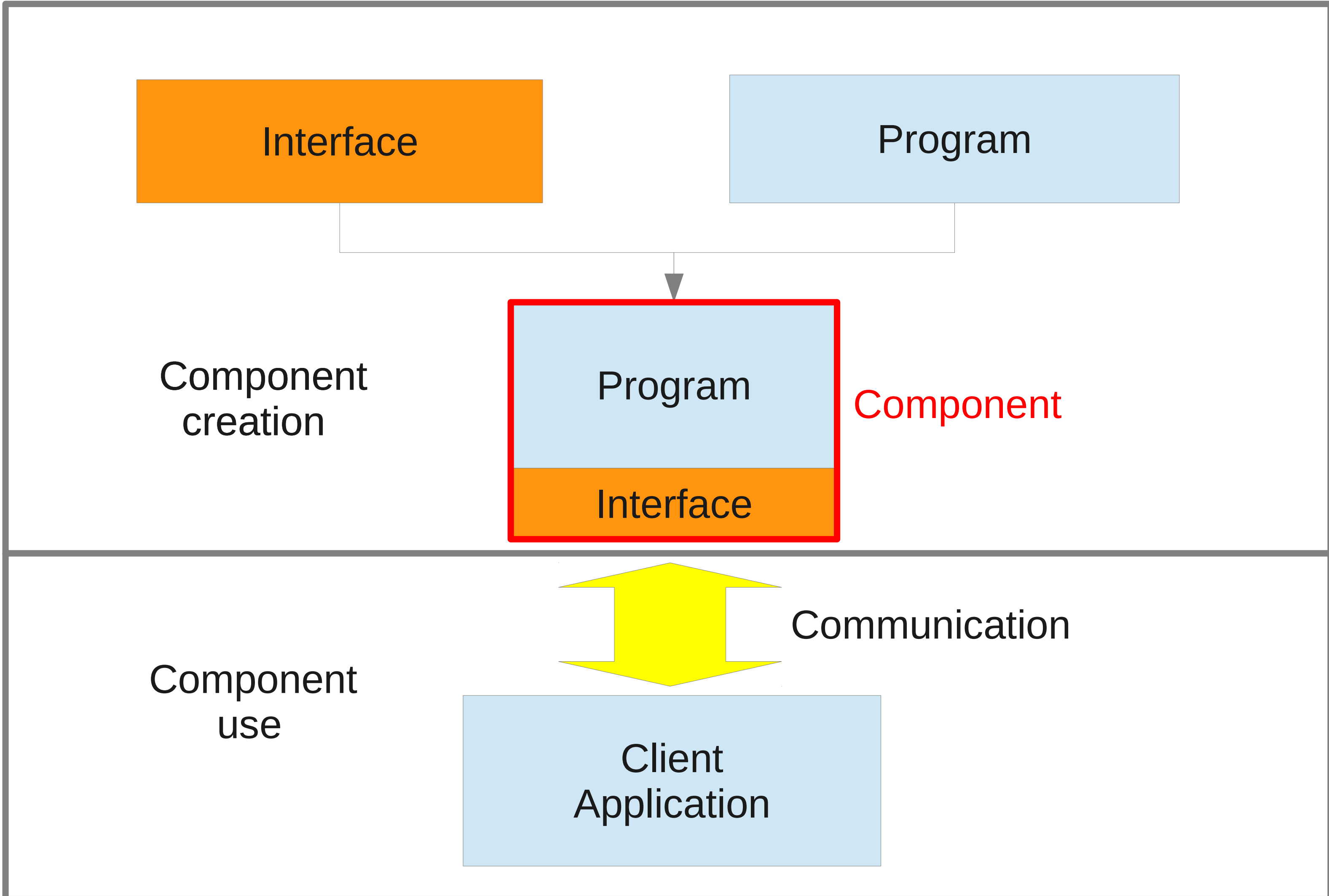
26.6.15 Braunschweig

CTL Introduction

Content:

- What is the CTL, what not?
- The concepts supported by CTL
- C++ representation of the concepts
- Examples of usage

Software Components



What is the CTL

The CTL is a light weight C^{++}-implementation of a minimal set of concepts allowing Component Based Software Developement (CBSE) and keeping design decisions in user hand

The CTL is not a domain specific implementation of coupled simulations but simplifies essentially there building up and distributed execution

The Supported Concepts I

- Static software component
- Run-time component = component object
- On demand instantiation of component objects
- Lifetime control of component objects by ownership relation
- Component object as server daemon
- Transparent linkage of component objects
- Component object groups supporting parallism

The Supported Concepts II

- Structured Component interfaces
  - Nested libraries
  - Optionally type paramerised global functions
  - Optionally type paramerised interface classes with inheritance
  - Use of type abstractions
- Asynchron remote calls
- Transportable remote objects with lifetime control
- Explicite and selective connection/binding of implementations to interfaces

Software Components

A (static) software component exists both as executable and as dynamic library, e.g. as Add.ctl.exe and Add.ctl.so.

Direct usage of the component:

```
Add.ctl.exe -I
```

list the implemented interface classes and functions

```
Add.ctl.exe -e <args>
```

run the embedded test with arguments *<args>*

```
Add.ctl.exe -l dmn 55555
```

start daemon listening at port 55555

Remark: on linux systems a .so can be made executable, .exe !needed

Components objects I

A component object (CO) is either

- the client process
- a daemon-process
- a thread or process started and owned by another CO
- In spite of daemons COs are instantiated on demand and their lifetime is controlled by ownership (creator is owner)

Components objects II

The instantiation of a component object is normally done in two steps

- Specify the address and linkage of the component, e.g.

```
location addLoc(„user@host:~/??/add.ctl.exe -l tcp")
```

- Instantiate the component and keep link to created CO for later usage

```
link addLink(addLoc);
```

Linkage Types I

Overview of the supported linkage types.

linkage	used tools	Comm. via	creation	connection
lib	libdl	function call	dlopen	dlsym
thread	libpthread	job-queue	dlopen/ pthread_create	dlsym
tcp	sockets/ssh	tcp/ip	ssh/spawn	connect/accept
pipe	pipes/ssh	stdin/stdout	ssh/spawn	Linux pipes
mpi	mpich/lammpi/ openmpi	MPI_Ibsend/ MPI_Recv	ssh/mpirun	MPI_Init
dynmpi	lam/openmpi	MPI_Ibsend/ MPI_Recv	MPI_spawn	MPI_Init
pvm	pvm3	pvm_send/ pvm_recv	pvm_spawn	pvm
dmn	sockets	tcp/ip	direct start	connect/accept
file	libstdio	read/write	fopen	fopen

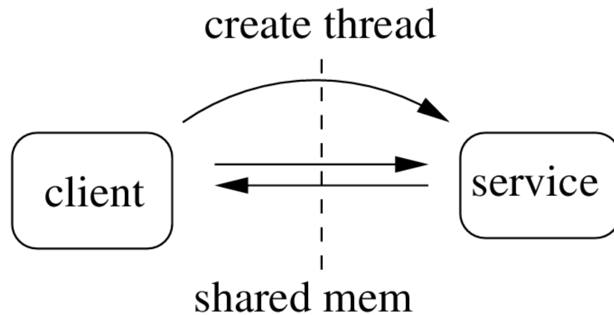
Only standard tool of linux/windows are needed.

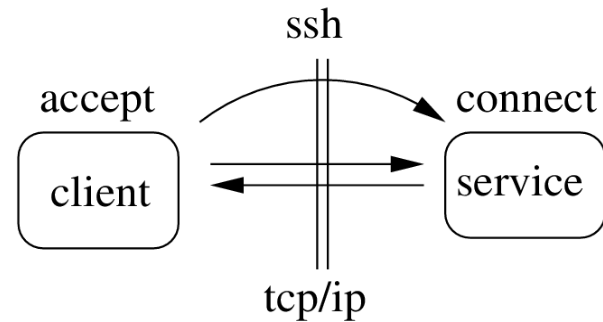
Linkage Types II

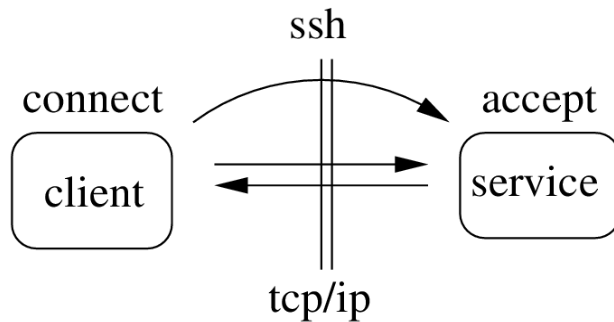
Overview of some properties of the linkage types.

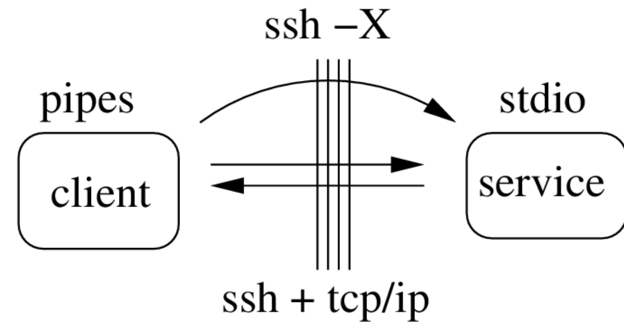
linkage	location	remote	as group
lib	path/lib -l lib	no	no
thread	path/lib -l thread	no	yes
tcp	user@host:path/exe -l tcp	yes	yes
pipe	user@host:path/exe -l pipe	yes	no
mpi	host:path/exe -l mpi	yes	yes
dynmpi	host:path/exe -l dynmpi	yes	yes
pvm	host:path/exe -l pvm	yes	yes
dmn	host:port	yes	no
file	libstdio	no	no

Linkage Types III

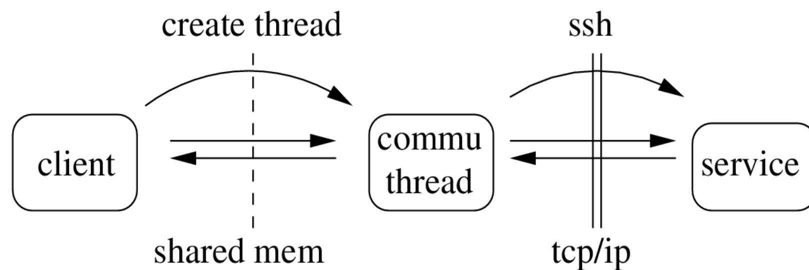




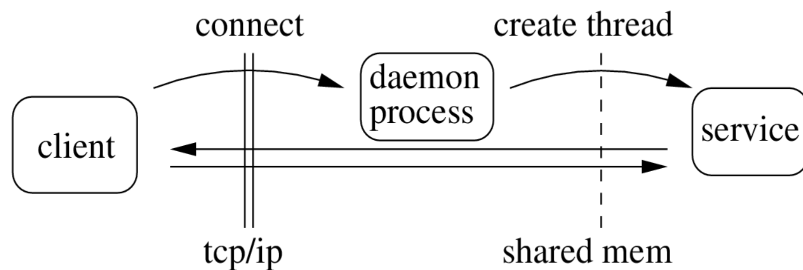




Linkage Types V



- Moves communication overhead into seperate thread



- Contacts daemon,
- Daemon creates service thread
- Service contacts the client

Component Interfaces I

- A component interface (CI) declares the functionality which may be available in a component object in form of functions and classes.
- Interface functions and classes can be type parametrised.
- Invocation of a non implemented function effects exception throwing
- If a class is available than all of its members

Component Interfaces II

Add.ci // compiler friendly

```
#define CTL_Class addCI
#include CTL_ClassBegin
# define CTL_Constructor1 (int4 /*param*/), 1
# define CTL_Method1 int4, add, (int4 const, int4 const) const, 2
# define CTL_Method1V add, (), 0
#include CTL_ClassEnd
```

Add.pci // user friendly

```
interface addCI
{
  addCI(int4 param);
  int4 add(int4 const, int4 const) const [];
};
```

Extended Component Interfaces

```
interface level1::derivedCI

interface multiplyByCI<type T, int N>
{
  int4 useOther(const level1::derivedCI);
  T | std::exception add (const T, T) const [];
  static T addS (const T, T) [];
  addV7   := add[7];
  T operator()(const T x, T y) const;
  {
       T z=x*y;
       y = addS(z, y);
       return x+y;
  }
11: aggr    := { add; useOther };
}
```

Connect Interface-Implementation I

connectAddCI.cpp

```
// tell compiler to generate the connecting code
#define CTL_Connect

// get in: interface and implementation
#include <ci/add.ci>
#include <add.h>

// called automatically in component instantiation phase
void CTL_connect()
{ ctl::connect<AddCI, Add>(); }
```

Connect Interface-Implementation II

If names differ or if there are ambiguities due to overloads than connect details must be specified:

```
class Plus {
...
  int plus(int, int) const {...}
};

struct connectDetail {
 CTL_Method(1, int, Plus::plus,(int,int) const, 2);
};

// change connector to:
void CTL_connect()
{ ctl::connect<AddCI, Plus, connectDetail>(); }
```

Asynchron Remote Calls

```
int val;

ctl::link addLink(addLoc); // creating component object

// create remote addCI object
addCI adder(addLink);      // creating remote object

// synchron call
val = adder.add(3, 5);

// asynchron call, 'res' acts as mailbox
ctl::result<int> res = Adder.add(3, 5);

// if answer arrived fetch the result
if(!!res)
  val = res;
```

Asynchron Remote Calls

```
int val;

ctl::link addLink(addLoc); // creating component object

// create remote addCI object
addCI adder(addLink);       // creating remote object

// synchron call
val = adder.add(3, 5);

// asynchron call, 'res' acts as mailbox
ctl::result<int> res = Adder.add(3, 5);

// if answer arrived fetch the result
if(!!res)
  val = res;
```

Remote objects/Remote Object References

- In each component an arbitrary number of remote objects (ROs) creatable via constructor of interface classes
- On service side references to ROs act as usual C++ objects
- References of ROs can be given as arguments of CI-functions to other COs
- Named remote objects enable referencing of the same RO by different CO's
- Lifetime control by global reference counting based on the credit method

Support of Parallel Applications I

A group represents a completely connected set/graph of component objects. The linkage of a group is uniform.

Creation of internal groups:

```
ctl::group("path/component -l <linkage> -n #proc")
```

where `linkage` in { `tcp`, `pvm`, `dynmpi`, `thread` }.

Creation of external groups:

```
ctl::link("path/component -l <linkage> -n #proc")
```

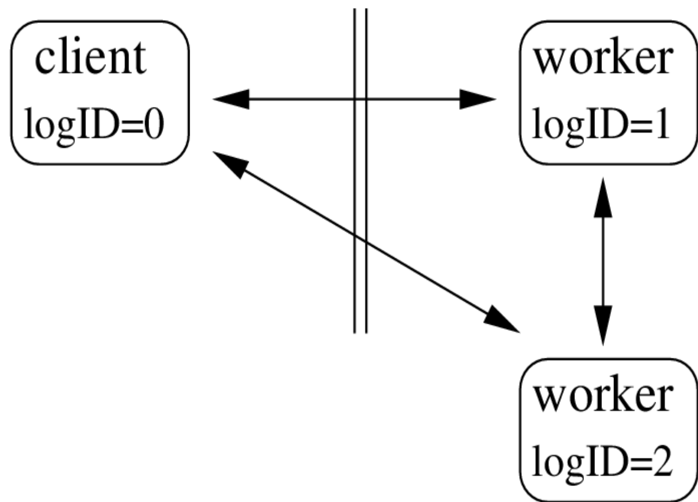
where `linkage` in { `tcp`, `pvm`, `dynmpi`, `thread` }

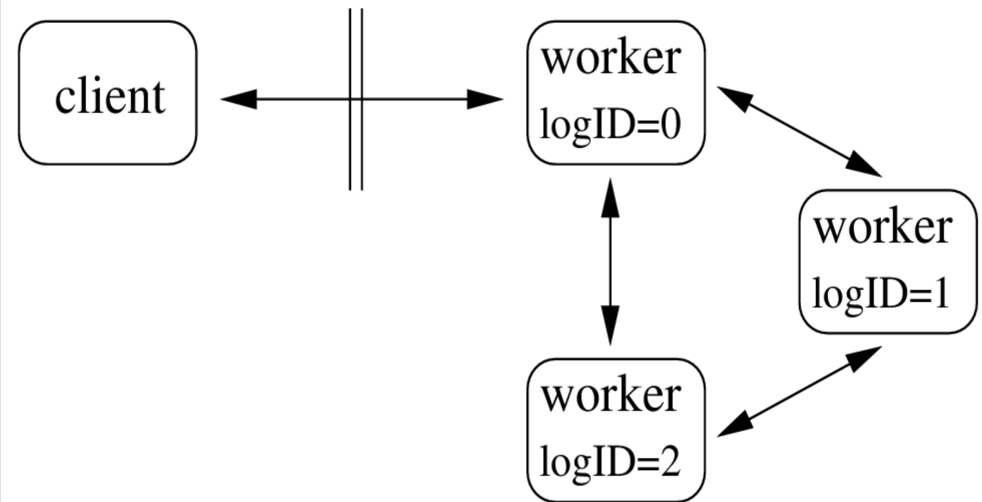
or

```
ctl::link("host@path/scheme -l <linkage>")
```

where `scheme` is a suitable application scheme.

Support of Parallel Applications II



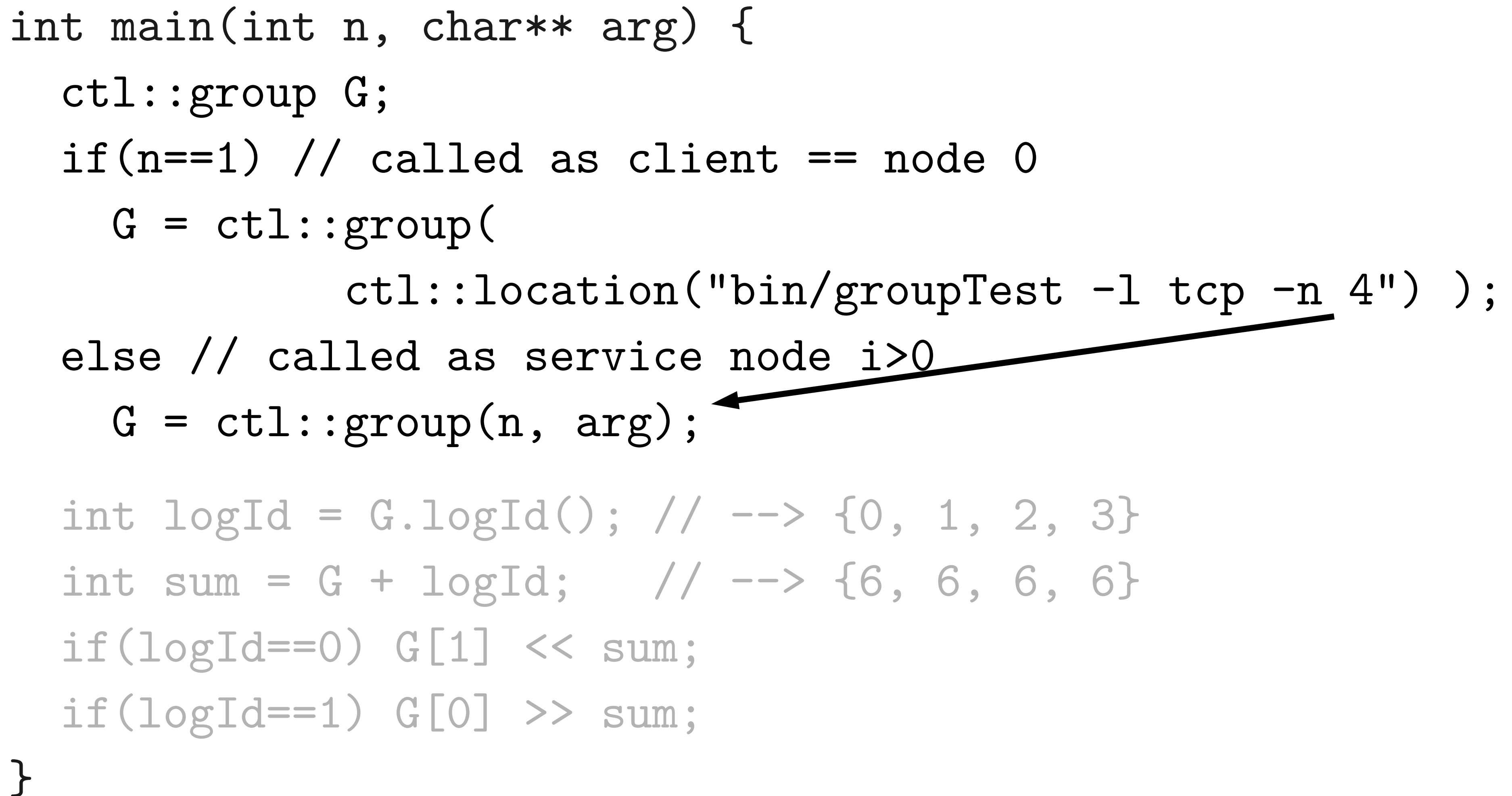



```
Effect of:
ctl::group("path/component -l tcp -n 2")
```

```
ctl::link("path/component -l tcp -n 3")
```

Support of Parallel Applications III

GroupTest.cpp with instatiating itself:



```
int main(int n, char** arg) {
  ctl::group G;
  if(n==1) // called as client == node 0
    G = ctl::group(
            ctl::location("bin/groupTest -l tcp -n 4") );
  else // called as service node i>0
    G = ctl::group(n, arg);

  int logId = G.logId(); // --> {0, 1, 2, 3}
  int sum = G + logId;   // --> {6, 6, 6, 6}
  if(logId==0) G[1] << sum;
  if(logId==1) G[0] >> sum;
}
```

Support of Parallel Applications III

GroupTest.cpp with tricky initialisation:

```
int main(int n, char** arg) {
  ctl::group G;
  if(n==1) // called as client == node 0
    G = ctl::group(
             ctl::location("bin/groupTest -l tcp -n 4") );
  else // called as service node i>0
    G = ctl::group(n, arg);

  int logId = G.logId(); // --> {0, 1, 2, 3, 4}
  int sum = G + logId;   // --> {10, 10, 10, 10, 10}
  if(logId==0) G[1] << sum;
  if(logId==1) G[0] >> sum;
}
```

Component Interfaces III

A set of basic interfaces is prefined

```
library functorCI
{
  interface namedCI<type Name, type CI>: CI;

  interface indexedCI<int n, type CI>: CI;

  interface unaryCI<type Y, type X>;
  { Y operator()(X) const [] };

  interface binaryCI<type Y, type X0, type X1>;
  { Y operator()(X0, X1) const [] };
  ...
}
```

Support of Parallel Applications IV

```
class gcdImpl {
  mutable int meM, nM;
  ctl::vector<gcdCI> teamM; // my team, used as ring
public:
  gcdImpl(const ctl::group &G, int n): meM(G.logId()), nM(n)
  { teamM = G.scatter(gcdCI(this)); }
  void operator() (int x) const {
    if(x!=nM)  { // terminating the recursion
      int s = teamM.size();
      nM = (std::max)(nM-x, x-nM);
      teamM[(meM+s-1)%s](nM); // call left
      teamM[(meM+1)%s](nM);   // & right neighbour in ring
    }
  }
};
```

Support of Parallel Applications IV

```
class gcdImpl {
  mutable int meM, nM;
  ctl::vector<gcdCI> teamM; // my team, used as ring
public:
  gcdImpl(const ctl::group &G, int n): meM(G.logId()), nM(n)
  { teamM = G.scatter(gcdCI(this)); }
  void operator() (int x) const {
    if(x!=nM)  { // terminating the recursion
      int s = teamM.size();
      nM = (std::max)(nM-x, x-nM);
      teamM[(meM+s-1)%s](nM); // call left
      teamM[(meM+1)%s](nM);   // & right neighbour in ring
    }
  }
};
```

Support of Parallel Applications V

```
typedef functorCI::unaryCI<void, const int> gcdCI;

int main(int n, char**arg) {
  ctl::group G;
  ...
  gcdImpl g(G, 6*(logId+1));
// create a gcdCI implemented by a reference to g
  gcdCI gci(&g);
// start gcd recursion but keep asynchron here!
  ctl::result<void> gres = gci(0);
// run until termination detected
  G.run();
}
```

Support of Parallel Applications V

```
typedef functorCI::unaryCI<void, const int> gcdCI;

int main(int n, char**arg) {
  ctl::group G;
  ...
  gcd g(G, 6*(logId+1));
// create a gcdCI implemented by a reference to g
  gcdCI gci(&g);
// start gcd recursion but keep asynchron here!
  ctl::result<void> gres = gci(0);
// run until termination detected
  G.run();
}
```

Support of Parallel Applications VI

Already in the parallel gcd example the determination of termination is non-trivial.

Solution: vector-method

- Keep book houlding of each task sent and received in each component for { 1,...,#procs}
- The book houlding vectors are accumulated by a „clever“ send-arround.

Support of Parallel Applications VII

```
struct anyFunctor {
  int operator()(double x) const;
} f; // thats our functor

typedef functorCI::unaryCI<int, const double> workerCI;

ctl::vector<int> test1(const ctl::vector<double>& x)
{
// hire on each thread one worker twice + one local worker
  ctl::vector<workerCI> worker(2*7+1);
  for(size_t i=0; i<7; i++)
    worker[i+7] = worker[i] = workerCI(f, ctl::thread);
  worker[2*7] = workerCI(f, ctl::lib);

  return ctl::evaluate<int>(worker, x, /*chunkSize*/10000);
}
```

Support of Parallel Applications VII

```
struct anyFunctor {
  int operator()(double x) const;
} f; // thats our functor

typedef functorCI::unaryCI<int, const double> workerCI;

ctl::vector<int> test1(const ctl::vector<double>& x)
{
// hire on each thread one worker twice + one local worker
  ctl::vector<workerCI> worker(2*7+1);
  for(size_t i=0; i<7; i++)
    worker[i+7] = worker[i] = workerCI(f, ctl::thread);
  worker[2*7] = workerCI(f, ctl::lib);

  return ctl::evaluate<int>(worker, x, /*chunkSize*/10000);
}
```

Fortran/Ansi C components

- Only interface classes
- Only one remote object per CO
- Only fundamental and array types
- Binding of methods by naming convention

→ see ctl/examples3/simu/fortran

Fortran connect

ConnectSimu.cpp:

```
// tell compiler to build fortran wrapper
#define CTL_ConnectF
// fortran subroutine names of the form: simu_<method>_impl
#define CTL_ClassPrefix simu

#include <ci/simu.ci>


void CTL_connect()
{ ctl::connectF<simuCI<double>, ctl::Extern::simuCI>(); }
```

Se also:

http://www.wire.tu-bs.de/forschung/projekte/ctl/e_ctl.html

---> CTL Manual p33ff

Usage by Python clients

Swig file needed like:

```
%module add
%{
#include "add.ci"
%}
%include "swig/add.h"
```

where swig/add.h is build by the makefile.
Python-C++ wrapper generated by SWIG code
→ see ctl/example1/add/swig

Python clients

Python client like:

```
import sys
sys.path.append( 'swig' )
from add import *
AddCI.use('bin/add.ctl.exe')
a = AddCI()
s_1 = 2; s_2 = 3
s = a.add(s_1, s_2)
```

http://www.wire.tu-bs.de/forschung/projekte/ctl/e_ctl.html
---> CTL and Python

Type Abstraction

Idea:

- any structured type is a composition of simpler types
- to read a structured data only its binary representation is needed

Fundamentals:

char = bool with values in {0,1}

char=int1, int2, int4, int8

unsigned char =uchar=uint1, uint2, uint4, uint8

real4, real8

Array [dim1=any][dim2=any]... :   T[dim1][ ]

```
template<class T> class array;
```

serialisation as

```
os<<int8(vec.size());
for(int8 i=0; i<vec.size; i++)
  os<<vec[i];
```

Examples of array<T>

- std::vector<T,Alloc>: array<T>
- std::set<T, Cmp, Alloc>: array<T>
- std::list<T, Alloc>: array<T>

Empty: {}

```
struct empty {};
oStream &operator<<(oStream &os,const empty&)
{ return os; }
```

Tupel: { t0, t1, … tmax }

```
template<class T0, class T1=empty,..., class Tmax=empty> class tupel;
{ T0 t0;
  T1 t1;
  …
Tmax tmax;
}
Serialisation:
os<<t0<<t1<<...<<tmax;
```

examples

```
std::complex<T> : tupel<T,T>
std::pair<S,T> : tupel<S,T>
map<key,val,Cmp,Alloc> : array<tupel<key,val> >
```

Thanks for your attention!