ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THP07

A MODERN C++ MULTIPOCESSING DOOCS CLIENT LIBRARY IMPLEMENTATION

S. Meykopff†, Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany

Abstract

At the DESY site in Hamburg/Germany the linear accelerators FLASH and European XFEL are successful operated by the control system DOOCS. DOOCS based on the client–server model and communicates with the matured SUN-RPC. The servers are built with a framework which consists of several C++ libraries. The clients use a DOOCS client library implementation in C++ or Java. In the past years the public interface (API) of the C++ client library was refined. But modern C++ features like futures are not provided in the API. Massive multi-processing, parallel communication, and optimized names resolution could improve the overall communication latency. The usage of the standard C++ library, the limit of external dependencies to ONC-RPC (former SUN-RPC) and OpenLDAP, and the reduction of the code size, increases the maintainability of the code. This contribution presents an experimental new client C++ library which achieves these goals.

BIRDS EYE ON DOOCS

The DOOCS control system uses the proven ONC-RPC (formerly SUN-RPC) for data transmission. To establish a connection to an ONC-RPC server, you need a host name and a port number. A DOOCS address must therefore first be resolved into a combination of host name and port number. This resolution is started via an LDAP query. If the server locations are served via multiple servers, all required information is available in the LDAP response. If all locations are answered via a single server, the existing locations can be queried via an RPC call to these servers.

Address Resolving with LDAP

The LDAP connection is established with OpenLDAP. First a connection to the LDAP server must be bound via ldap_simple_bind_s(). The single queries are then made with ldap_search_ext_s(). Depending on which entries are searched for, the distinguished name (DN) must be created at runtime. The DN consists of the following Relative Distinguished Names (RDN) "location", "device", "facility", "dc". The filter consists of a string with a "server-mask" and partly "device" as logical link. The last parameter to be specified is the scope in which to search. This also depends on which DOOCS address part is searched for. The connection to the LDAP server can be reused until all address resolutions are finished. At the latest at the end of the program the connection must be closed with ldap_unbind(). A sequence diagram is shown in Figure 1.

LDAP Answer

The LDAP query response contains these fields: "facility", "device", "location", "property", "server", "host", "channel", "protocol", "lib-prog", "server-mask", "status". The first four are part of the DOOCS address. The fields "host" and "lib-prog" are needed to connect to a DOOCS server. The two fields contain the above mentioned connect information for the RPC call. When searching for locations, the "property" field remains empty.

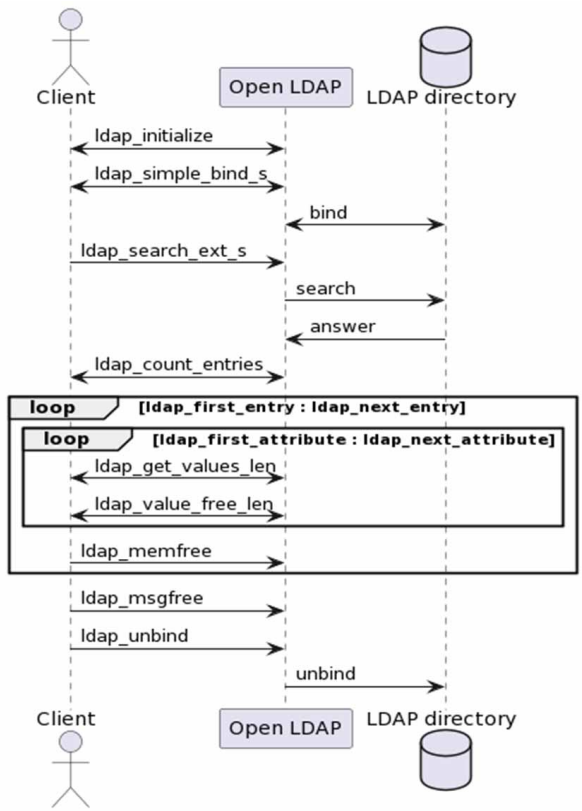

Figure 1: Address Resolving.

Authorization

To establish an RPC connection, authorization must first be performed. The authorization is provided by the authunix_create() function. The function requires the target host name, the (Unix) user ID and the group ID of the person to be logged in as parameters. The authorization is valid for all threads of a task until the auth_destroy() function deletes it again.

RPC Bind

A connection to the target server will be opened with clnt_create(). The function needs the host name as parameter and the port number determined above as parameter. The choice whether UDP or TCP is used as protocol must now be made. However, UDP is not used in DOOCS because the RPC messages may otherwise have a maximum of 8 kbytes of coded size. The function expects the protocol type as string. After the connection has been established,

† sascha.meykpoff@desy.de

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THP07

the function returns a client pointer. The authorization created above must now be entered in the cl_auth field. From now on the connection can be used for communication with RPC messages.

Serialization with XDR

The core of RPC communication is a function call on another system. Not only must a function be started over the network, but function parameters or return value must also be transferred. These parameters must be serialized. Since the foreign system or also the transmission layer can use another byte order, the serialization must consider this also. ONC-RPC describes the parameters in its own language. An RPC specification (XDR) contains a number of definitions. The language supports scalar data types, arrays, constants, enumeration, struct, and union. The parameter description is normally stored in .x files. The program rpcgen creates from the .x files client and server stubs in C. To remain compatible to DOOCS the original DOOCS source files are used. The DOOCS library encapsulates the data in EqData objects. These simplify the copying of the data considerably. For simplicity, these EqData objects are also used for this project.

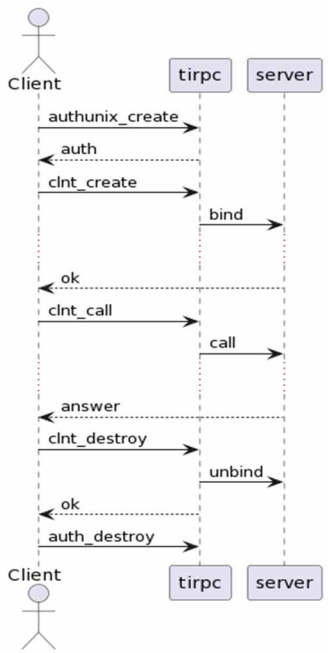

Figure 2: RPC Sequence Diagram.

Remote Call

To do a data exchange, the ONC-RPC function clnt_call() is called. The function expects the client handle, the proc number and a timeout. The proc number is an enumeration on get, set, or get_name. A pointer to a param_block structure describes the data to be sent. The structure consists of a pointer to an EqNameString structure and a pointer to an EqDataBlock structure. In the EqNameString the parts of the destination DOOCS address must be entered. The EqDataBlock structure is the fundamental DOOCS data container. This container consists beside some timestamps and error codes mainly of the DataUnion structure. The DataUnion structure has the field data_sel, which specifies the DOOCS data type. In the following union all DOOCS data types are described. How the union is evaluated depends on the data_sel field. Graphic 2 provides an overview.

Return Value and Cleanup

How the structure must be serialized is communicated to the clnt_call() function via a pointer to the data generated with rpcgen. The return value of the RPC call is written to an EqDataBlock structure which is also passed. This structure must also be described by a pointer on an RPC specification. The clnt_call() call is blocking. It returns a clnt_stat parameter as status, which is RPC_SUCCESS if successful. The EqDataBlock structure of the return parameter is allocated with malloc by the clnt_call() function. If the structure is no longer needed, it must be freed using clnt_freeres(). An unused connection is closed with clnt_destroy().

Receive Location Information

If you need to retrieve the list of locations or properties from a server, this is done via a modified RPC call. Instead of the proc number for get, the enumeration value for get_names is used. The location and the property name parts of the DOOCS address remain empty. The response from the server is a list of USTR data types. The USTR type contains among other things a string. Here the location name is stored.

Receive Location Information

In the following, a use case is considered where different control system values are read in simultaneously during a measurement. The classic DOOCS API makes several synchronous blocking RPC calls here. Here the addresses are resolved one after the other and then the blocking clnt_call is called in each case. A full sequence is shown in Fig. 2.

OPTIMIZATION

The first optimization concerns the API. If the data is returned directly in the API, it is also necessary to wait until the data has been transported completely. The experimental API returns only one std::future. In the API a thread serves the corresponding std::promise. If the data has arrived, the method set_value() of the promise is called. The method set_exception() is used in an error case. The client can wait using the wait() methods of the future until the data arrived. In the meantime, the client can start other requests, or do something else. If the wait function signal returns, the client can fetch the data with the get() method. In the best case the client waits now maximally for the time of the slowest call.

Parallel LDAP Calls

By simply replicating an LDAP database, several servers can easily be set up in parallel. Normally, these are backup servers that step in when one server is unavailable. An optimization is obvious at this point. The LDAP queries

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THP07

put into a queue. A thread pool then processes the queries in parallel. A separate thread is used for each LDAP server. This allows multiple LDAP servers to be used simultaneously. A connection to the LDAP server should not be terminated immediately. Since it cannot be predicted when the next LDAP query will occur. A timer should terminate unused connections after a defined time.

Parallel RPC Calls

The RPC call can also run in parallel. In this case, communication with each server takes place via a separate thread. To conserve the resources of a client system, an upper limit for threads should be observed. In this case, a thread must process several connections. More than one connection to a server must be prevented at the moment, since it is not yet known whether the servers are internally multi-threaded. Also, the RPC connections are not closed after the last access. The slow connection setup must otherwise be restarted for future queries. A timer must also close unused connections after a certain time. A full sequence of two parallel calls is shown in Figure 3.

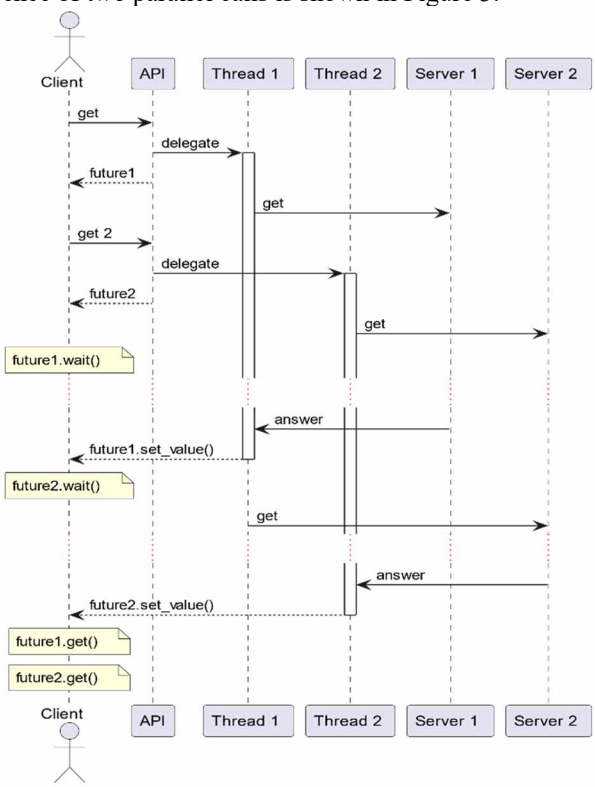

Figure 3: RPC Optimization.

IMPLEMENTATION

At the beginning, the API shall only support an asynchronous query of DOOCS properties. The method call in the RPC_API object is defined as follows:

```
std::future<std::unique_ptr<EqData>> read(
    const std::string &doocs_address,
    EqData *send_data
);
```

The doocs_address parameter contains the DOOCS address in the complete form ("FAC/DEV/LOC/PROP") and is separated internally into the individual address parts. The EqData parameter is a pointer. So that the passed data can be released or processed immediately after the end of the function, the EqData data block must be saved internally. The return value is a std::future with a std::unique_ptr on an EqData data block as described above. The scope of the internally generated EqData block is shifted to the user code by the unique_ptr. If the user does not need the block any more, it is released immediately. The proceeding should prevent memory holes. The use of EqData is purely a convenience decision. The EqData object provides easy access to all data types and also handles the copying of data.

RPC_INFO Object

If the API works with std::future, a std::promise must be managed internally. The std::promise must not be answered until the data request is finished. The promise is saved with other information such as the parts of the DOOCS address, data to send, timeout information, host server names and port numbers. On the first call in the experimental API the necessary information is copied into a rpc_info object. The rpc_info object can now be moved to the individual processing queues. At the end of the processing chain the promise is answered and the rpc_info object will be freed.

LDAP Cache

The already resolved LDAP requests are stored in a ldap_cache class. A std::unorderd_map holds the data and a mutex provides thread safety. Still within the get() method of the rpc_api class, the ldap_cache is searched for an entry. If none is available yet, the rpc_info object is put into the queue for LDAP query.

LDAP Queue

The ldap_queue is a std::queue which takes the rpc_info object. A thread pool get the data from the queue and starts the LDAP requests. Successful requests are recorded in the ldap_cache. For this purpose, a std::function object is defined in the rpc_info object. The function references the ldap_cache object and will be used as a callback. Thus the ldap_queue has no dependency to ldap_cache. Testing the ldap_queue is made easier by having fewer dependencies. If the LDAP query is not yet sufficient the server is queried via RPC. The rpc_info object is pushed into the RPC queue described later. Once the name resolution is complete, a second std::function is used as a callback back into rpc_api. In the scope of rpc_api, the RPC_INFO structure is moved into the rpc_scheduler.

RPC Scheduler

The dispatch() method of the rpc_scheduler object receives the rpc_info object. The rpc_scheduler has a rpc_dispatch object for each RPC connection. The rpc_dispatch object consists of a thread and a rpc_info queue. If no rpc_dispatch exists for the RPC connection, a new rpc_dispatch object is created. In the constructor of rpc_dispatch a rpc_auth object is necessary. Later the

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THP07

rpc_auto object will authorize the communication. The dispatcher now moves the rpc_info data into the rpc_dispatch object. The thread of rpc_dispatch will take an entry from the queue and continue with the processing. For the RPC call a connection to the server must exist. Therefore rpc_dispatch has a rpc_connection_map object. The object stores each connection in an unordered_map. This is important when rpc_dispatch needs to make queries to multiple servers. A centralized map is not possible because in the current ONC-RPC implementation each thread must establish its own connection. The rpc_connection_map object now creates a server connection and also terminates it when the rpc_connection_map is cleaned up. The RPC handle can now be read from the rpc_connection_map object and the RPC call can be performed. Once this is finished, the res_block structure is transformed into an EqData object and copied into a std::unique_ptr<EqData>. This will be moved into the callback. The callback ends up in a method in the rpc_api object. The rpc_api scope moves the value into the std::promise. Now the rpc_info object is released and the query is finished.

MEASUREMENT

For the operation of the machines European XFEL and FLASH mainly JDDD is used. JDDD structures the data in so-called panels. The data read from the control system is displayed there. During a shift handover at the machines, screenshots are printed from the main panels into the electronic logbook. In total, there are about 30 screenshots. To get a valid collection of DOOCS addresses, these panels were opened simultaneously and the list of currently used DOOCS addresses were written to a file. The list contained about over 12000 entries. Duplicate entries, addresses of foreign control systems, one particularly slow server, and unreachable addresses were removed from the list, leaving just over 2800 usable DOOCS addresses pointing to 287 servers.

Timing Results

By getting the data from all addresses the runtime of the classic DOOCS library and the experimental library was measured. The results are shown in Table 1.

Table 1: Runtime Results

Code	Runtime	No. Threads
Classic	280s +-34s	1
Exp.	13s +- 1s	100
Exp.	13s +- 2s	50
Exp.	31s +- 8s	25
Exp.	64s +- 11s	10
Exp.	105s +- 12s	5
Exp.	143s +- 11s	2
Exp.	170s +- 21s	1

Figure 4 shows when the individual operations of the Classic DOOCS library are finished. About 200 queries require a total of 90s. A peak of 6 seconds (not plotted) for one call was measured. These slow responses are the lower limit for the experimental library.

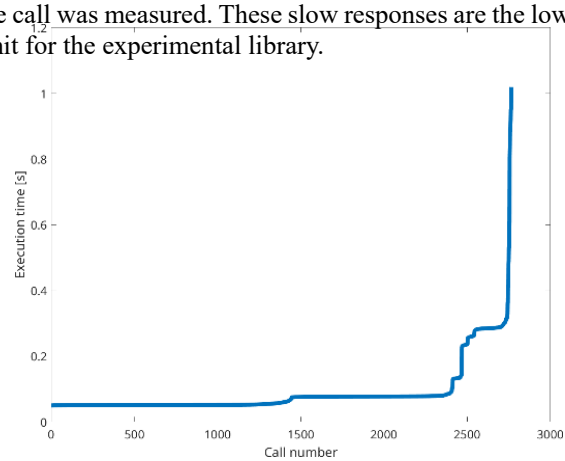

Figure 4: Runtime of finished RPCs.

Runtime Histogram

In the experimental library case Figure 5 shows a histogram when (since program start) how many queries have been completed. The most requests were finished after 10 seconds.

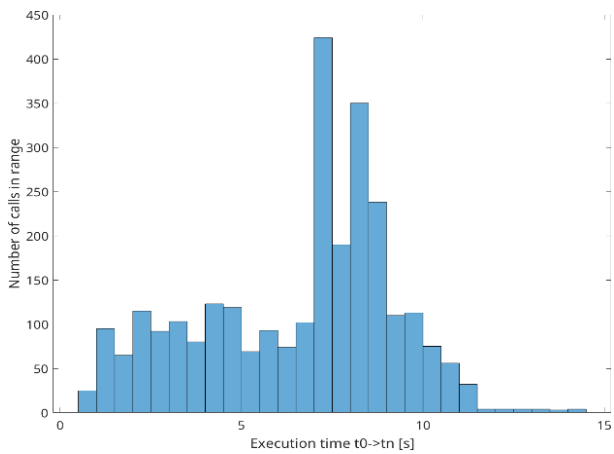

Figure 5: Runtime histogram.

CONCLUSION

In conclusion, the experimental library saves a considerable amount of time in this use case. The fact that the connections are not closed immediately reduces the overhead. The use of std::future with parallel communication corresponds to modern hardware utilization. The classic DOOCS library also offers asynchronous connections with "monitors". However, these connections are not intended for one-time communication. The experimental library offers a similar function. However, this functionality needs to be better implemented. Therefore, it has not been discussed in this paper. When planning the objects, it is a good idea to implement them independently as possible. This way, automatic tests can be used to ensure the quality of the software. The amount of work time required for such a project should not be underestimated. If it is not really necessary, it is always recommended to use already finished software.