TIME ACCURATE FLUID-STRUCTURE COUPLING EMPLOYING A LIGHTWEIGHT SOCKET BASED DATA EXCHANGE

Julius Klauck **Manuel Keßler**
University of Stuttgart
Institute of Aerodynamics and Gasdynamics
Stuttgart, Germany

ABSTRACT

A software library used for the fluid-structure coupling is presented. The library is designed to efficiently exchange data between two programs and uses a socket-based approach. Three different communication protocols are implemented. A synthetic test case is investigated to benchmark the performance of the library. It is shown that the socket based approach significantly increase the bandwidth of the communication compared to a file based approach. Afterwards a strong coupling scenario is presented where the library is used to couple the flow solver FLOWer to the multi-body-dynamics code SIMPACK. The results obtained are more realistic and the choice of protocol shows no significant impact on the overall simulation time. However, the library significantly reduces the programming effort for a new coupling framework and facilitates the entire coupling exchange with few API calls.

1. NOTATION

Acronyms:

API	Application Programming Interface
CFD	Computational Fluid Dynamics
ERF	European Rotorcraft Forum
HPC	High Performance Computing
IAG	Institute of Aerodynamics and Gasdynamics
MPI	Message Passing Interface

2. INTRODUCTION

Fluid-structure interactions are crucial for the numerical analysis of helicopter performance. Due to the need for a lightweight construction, rotor blades in particular can exhibit large elastic deformations. These deformations significantly influence the aerodynamics, which in turn affect the structure. For a complete understanding of the helicopter aeromechanics, this interaction must be included in any numerical investigations as shown in Ref. 1.

The general approach for helicopter applications is to have flight mechanics, structure deformation and aerodynamics bundled in a single program. Such a program is generally called a comprehensive rotorcraft tool of which CAMRADII Ref. 2 or HOST Ref. 3 are some examples. A major shortcoming of such tools is the simplified aerodynamic model employed. To improve the accuracy, the aerodynamics can be modified from results obtained via hight fidelity computational fluid dynamics (CFD). This approach allows for specialized programs which can efficiently use the computational resources available, i.e. the CFD runs on a highly parallel cluster, whereas the comprehensive rotorcraft tool can run on a workstation. To simulate the fluid-structure interactions, aerodynamic loads and structural deformations have to be interchanged by these programs.

This method of coupling the comprehensive tool to a detailed CFD simulation is considered state of the art and was used at the Institute of Aerodynamics and Gasdynamics (IAG) to analyze complex interaction phenomena on helicopters, see Refs. 4 and 5. However, due to the high computational effort needed for a full helicopter CFD analysis, most investigations restrict themselves to the trimmed, periodic conditions of stationary flights. In this case, the so-called weak coupling approach, where periodic loads and deformations are exchanged, can be used. This means that the CFD and the comprehensive code run sequentially and obtain the coupling information from the other tool by interpolating the periodic data to their respective simulation time window. A general descrip-

tion of the strategy can be found in Ref. 6. This approach thus inherently enforces a periodic solution in the flow state as well as the structural domain, which in turn means that the investigated effects are mapped to the same base period. For helicopter applications the base period of the periodic solution is usually the time it takes for one full main rotor revolution, i.e. the reciprocal of the rotational frequency of the main rotor.

This technique of applying a weakly-coupled CFD is entering earlier stages in the helicopter design phases, see Ref. 7. Research now is moving towards the effects of unsteady helicopter motions on their performance and stability. Studies, where an unsteady maneuver was approximated by short steady segments were performed in Ref. 8. This approach is still limited by the common base period necessary for the weak-coupling approach and is thus not able to predict the impact of quickly varying pilot inputs or unsteady environmental conditions.

To fully capture these unsteady effects, the CFD and the flight/structure-mechanics code have to run in parallel and need to exchange data at every time step. This coupling setup is called ”strong” or ”tight” coupling as described in Ref. 9. A major disadvantage of this approach is the high data exchange frequency. An explicit coupling scheme needs at least two exchanges per coupling time step. The easiest and probably most widely used approach for data exchange between two programs is file based. Here, data is written to a file by one program, which is then read by the other program. These coupling files are usually human readable and not in a binary format, which adds additional overhead of converting especially numerical values. If both programs run on the same hardware, this file exchange is fairly straight forward, but for distributed hardware an additional file transfer becomes necessary. These file system accesses are a comparatively slow process compared to memory access which inevitably will slow down the overall simulation time and adversely affect parallel efficiency, as the processes need to wait longer until the data is transferred.

To ameliorate the effectiveness of strong coupling, the file system access can be replaced. This is achieved by accessing the computer memory directly. One approach is the use of so-called `sockets`. The socket-based coupling approach is not new and has already been used in Ref. 10. Many of the approaches that employ sockets as a means of data transfer are, however, highly specialized for their use in the respective codes. The aim for the work described in this paper was to create a lightweight, capable and most of all easy to use software library which handles only the data transfer between different programs. In the following research the general structure of the library is discussed and the performance of the socket-based data exchange is investigated by coupling FLOWer to the multi-body-dynamics code SIMPACK. With the improved performance it will be possible to investigate the unsteady effects of helicopter maneuvers on the aerodynamics and the structure in a more detailed manner.

For completeness it must be mentioned that there already exist some libraries which handle the fluid structure coupling task with multiple codes, which comprise an entire framework including mesh mapping, interpolation, communication, pre- and post processing. One example of such a library is preCICE which is used in Ref. 11. These coupling libraries need specific interfaces for the fluid as well as the structural dynamics solvers, enabling the communication between the different programs. Writing these interfaces can be a time consuming task, especially if there are already functions for structure coupling present in the codes.

3. METHODS

For helicopter applications at the IAG a weak-coupling framework is already in place, which handles the force calculation and interpolation of the data. Therefore, a simple data transfer is sufficient to increase the capability towards strong coupling. To handle this data transfer, a lightweight software library was developed by the helicopter group of the IAG, which is called `heliKomm`.

The main goal was to minimize the data transfer time to enable full helicopter maneuver simulations. Another important aspect was is the provision of a simple Application Programming Interface (API) to reduce the programming effort for any new coupling framework. As the library functions are bundled into one place and implemented only once, the test coverage of the individual functions is increased and consequently the number of bugs present reduced. As the library is used in different programs, it is guaranteed that the communication will work when the same versions of the library are used due to identical data structures and protocols.

Although heliKomm should be lightweight, a set of requirements must still be fulfilled so it can be reliably used:

1. A universal data protocol must exist to encode and decode the messages sent from one program to the other.
2. Different communication phases must be handled so that data can be exchanged at different locations in the respective programs.
3. It cannot be guaranteed that the message order on the distributed machines is identical. This must be addressed to prevent a dead-lock, i.e. both programs wait indefinitely for each other.
4. The byte ordering, i.e. little or big endian, on the distributed machines can be different.
5. Computational resources on High Performance Computing (HPC) clusters are still expensive. To prevent significant wait times due to crashed programs, a timeout mechanism must be included.
6. For security reasons a whitelist should be included, which allows only the listed machines to connect to the heliKomm instance.

For the communication between programs a socket-based approach was chosen. Most documentation on sockets can be found online. A brief overview on what sockets are and how they work is given in Ref. 12. For this paper the description of the functions does not include all implementation details of the socket communication as those are not necessary to understand the concepts for the communication. A socket can in effect simply be interpreted as a file where one can read and write data. The operating system then takes over and delivers the data to the corresponding communication endpoint. heliKomm implements three different socket protocols:

1. UNIX: This socket type is used for communication on the same machine and is based on direct memory transfer.
2. TCP: This socket type is used for communication over the internet via the IPv4.
3. FILE: This protocol uses files in a binary data format, which are written to disc and does not directly use the socket mechanism provided by the operating system. The protocol can be used directly for communication on the same file system. It can also be used for remote communication by using a mounted file system or by implementing a separate file transfer. The FILE protocol should serve only as a fallback if there are permission restrictions on creating sockets. To reduce the workload on the file system, it is only checked every 100 ms if a file is present, which reduces the bandwidth when writing many small files.

Information on the UNIX and TCP protocols can be found in Refs. 13 and 14. The UNIX and TCP protocols provide mechanisms for the programs to detect if a communication peer is available to connect to and also if a previously established socket connection was closed by the peer. This can be used to detect if a problem occurred during the runtime of the other program and corresponding action can be taken. The FILE protocol does not provide this functionality.

3.1 Implementation details of the library

3.1.1 Message protocol

A socket can only transfer data byte-wise. Therefore, a message has to be serialized into an array of bytes before it can be sent. For this reason a message protocol has been implemented to ensure that this serialization is possible. It is noted that all three socket protocols used here guarantee that all bytes are transferred and keep their respective order, which is a prerequisite for this protocol to function correctly.

The way the serialization of the message works in this protocol is rather simple. The first four bytes of the message encode the length of the message in bytes. This is a dynamic value and can change with the messages sent. The next byte encodes the version of the message protocol to check if the received messages match the protocol. This prevents obscure bugs if the message protocol is ever changed and communication still occurs with an older version. The next 64 bytes are reserved for a header string of the message. This is used to uniquely identify the message later. If the header size is changed, the version must be adapted as this value is hard-coded in the protocol. The rest of the message is reserved for the payload, which can be zero bytes long. Due to the way the protocol operates, there must be no other data sent in between messages.

The payload encodes its size as four bytes and its data type as one byte of which there are: CHAR, INT16, INT32, INT64, FLOAT, DOUBLE. The data stream then follows these five bytes. The payload size is the number of elements contained in the data, not the number of bytes. This is different than the message but simplifies using the payloads.

One notices that the payload size is encoded again, which one can deduce from the overall message size. However, the idea is to keep the payload and message separated as one can add specific payload types later if they become necessary.

3.1.2 Library Structure

The library aims to be universally usable, especially with distributed HPC systems. `heliKomm` uses the Message Passing Interface (MPI) Ref. 15, which is widely used by HPC software to enable different processes to communicate between each other. An MPI program works by starting the same program on multiple processor cores, which are referred to as ”ranks” in this paper. Depending on their rank number, different but similar tasks are performed by the processes, e.g. the process with rank number 0 is usually referred to as the master rank, which handles such tasks as the synchronization. All ranks are able to send data between each other. This MPI communication, however, usually only works inside of a single program. To overcome this limitation `heliKomm` exists to enable the communication outside of the program. The idea of the inter-program communication is as follows: the master rank of the program also handles the socket communication established by `heliKomm`. The data which has to be communicated between programs can be located on different ranks. Before this data can be transferred to another program it must be gathered on the master rank first, which then sends the messages over the communication sockets. `heliKomm` is also able to receive data from another program. The data is also received over the socket by the master rank and are then distributed to the ranks which require it. To handle the distributed data layout, a special logic and structure with groups and members is used. This not only enables the efficient inter-process communication but also allows the communication to be split up into different steps. This is necessary when some computations have to be performed between communications. In the case of fluid-structure coupling the CFD code first needs the deformation information from the comprehensive rotorcraft tool, with which the surface can be deformed and new loads calculated. By splitting the receiving task and the sending task into two distinct groups, a simulation phase can be done in between them.

The structure of `heliKomm` is split into three parts which handle different tasks:

1. The **Communication Manager**: A class which handles the API of the library and all groups. The Communication Manager of the master rank also handles the communication with the outside. A Communication Manager object is present on all MPI ranks.
2. The **Communication Group**: A Communication Group handles all the communication requests posted by the program. There can be multiple groups per Communication Manager. All MPI ranks must know about all groups but must not necessarily be part of any group.
3. The **Communication Member**: A Communication Member manages the messages. The member is always attached to a Communication Group but not all MPI ranks of a Group must necessarily contain all members of the group.

The following chapters give a detailed explanation of the workings of the three parts.

3.1.3 Communication Manager

The Communication Manager is the main API object. There can be multiple managers present in one program but one manager can only connect to one Communication Manager of a different program. To initialize a communication, two methods can be used. Firstly, the necessary information can be passed to the manager directly in the program. Secondly an input file, which contains all necessary information, can be read.

To establish a connection between two `heliKomm` instances one instance must be specified as the server and the other as the client. This distinction is only necessary to establish the connection. During the communication phases both instances are equal and can send and receive data, albeit the order these operations are performed in are switched for server and client respectively.

In case of the `FILE` protocol, server and client are handled identically. Both need to specify the path were the file is created and a file extension which links the file to both instances. Using different file extensions,communication with different Communication Managers can take place in the same folder. No actual connection between the two instances takes place when using the `FILE` protocol, it is rather only in the communication phase were it is checked if a file created by the communication partner is present.

For the `UNIX` protocol, a directory on the file system, a name for the socket and a port number must be given.

The Communication Manager on the server side creates the socket at the specified location and waits for incoming connections. Afterwards the client is able to connect to tp the corresponding socket. Server and client both wait until this connection is established. To prevent indefinite wait time, a timeout mechanism exists for both server and client side. The server simply waits a specified amount of time for a connection and aborts the process if none occurs. The client on the other hand makes multiple connection attempts during the specified timeout time until it is either successful or not. When the connection is successfully established both client and server can read or write data to the socket.

For the `TCP` protocol a network address and a port number must be specified. The protocol specifies a wildcard address ”0.0.0.0” to accept connections from any incoming IP address, which should be used by the server. The port number can be chosen freely as long as it is not already in use, albeit with some limitations, i.e., some specific port numbers are reserved for well established services. The server then creates the `TCP` socket and waits for an incoming connection. The client must specify the `IPv4` address or alternatively the domain name, e.g. ”pc.iag.uni-stuttgart” of the server. The client can then try to connect to the server. As for the `UNIX` protocol, the timeout mechanisms for client and server are also available. Furthermore, the server can specify a list of allowed connections peers (IPv4 addresses or domain names) where connection to all others is then denied.

After the connection to the other `heliKomm` instance is established data can be exchanged by the two programs. The Communication Manager implements the functions to handle this data transfer over the sockets. At this stage of the communication the data which must be sent is already converted to a `heliKomm` message, which can be encoded to a byte-stream as defined by the Message Protocol, see section 3.1.1. This byte-stream is written entirely to the socket, where the operating system delivers it to the communication endpoint. To receive a message from the socket the Message protocol is used to decode the byte-stream coming from the socket. As the first four bytes in the stream encode the message length, this many bytes are read from the socket if they are available. A timeout mechanism exists for both reading to and writing from the socket. This ensures that the program is stopped if one `heliKomm` instance closes during the program runtime. All messages are decoded in the order they are received from the socket. For that matter, received messages can be temporarily stored if they are not immediately required and the next message is read from the socket. Consequently it is checked first if a messages is already stored before a new one is read from the socket.

The available API calls of the Communication Manager are depicted in Figure 1, which also puts them in the order they are used in a program. A brief list of the those functions with a short description and the ranks which must call them is given in Table 2.

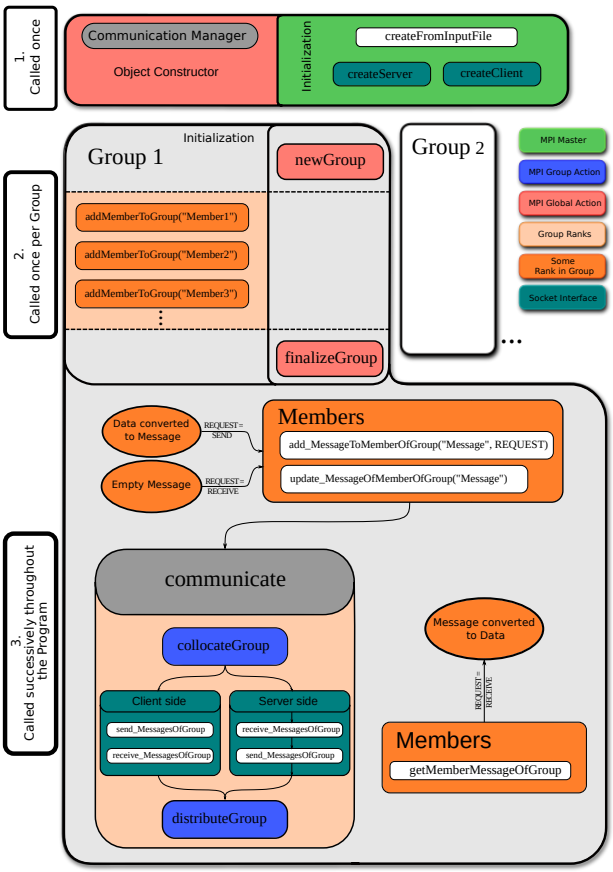

Figure 1. Overview of the API calls and the place in the program where they can be called.

3.1.4 Communication Groups and Members

As mentioned previously in section 3.1.2, the Communication Groups are necessary to handle the MPI functionalities of `heliKomm`. Data can be distributed over multiple ranks, whereas only the master establishes a socket connection. Therefore, the data has to be gathered on this master rank first to be sent over the socket. Likewise, any data that is received must be subsequently distributed to the other MPI ranks again.

To establish this group logic, the group must be created with the `createGroup()` API call first. This must be called by every rank for every group even if the rank is later on not part of this group. Afterwards, group members can be added to this group with the `addMemberToGroup()` call. Multiple members can be part of the same group. If for example one wants to investigate the deformation of a ro-

tor, every rotor blade can be added as a different member of the group. For a usual CFD code the fluid volume is distributed over multiple MPI ranks. Therefore, only those ranks which handle the corresponding rotor blade need this deformation information and must be part of the group and call `addMemberToGroup()`. The master rank is implicitly added to all groups as a member as it is responsible for the socket communication and must collate and distribute the data later on. To complete the group creation `finalizeGroup()` is called. This call must be made by all MPI ranks in order for the master rank to have a list of the ranks which are part of the group. After the group is finalized members cannot be added or removed from the group anymore.

3.1.5 Communication phase

With the group logic in place `heliKomm` can now be used to communicate over the sockets. The main function to handle the communication is the `communicate()` call. Before `communicate()` is called the messages which will be sent or received must be registered to the Communication Members. This is done with the `add_MessageToMemberOfGroup()` API call. The `heliKomm` library provides functions to convert data of different types, e.g. a text string or an array of floating point numbers, to a message object which conforms to the Message Protocol. As all messages are sent over the same socket, the message headers which identify them could conflict. Therefore the message header is prefixed with the group and member name when the message is added to the member. The master rank must know if the message should be sent over the socket or if one should be received. This is handled in the `add_MessageToMemberOfGroup()` call by providing a communication request. There exist five types of communication requests, two for sending, two for receiving messages and one empty `NONE` request. The requests are: `SEND`, `SEND_ONCE`, `RECEIVE` and `RECEIVE_ONCE`. The `SEND` or `RECEIVE` request are handled every time the `communicate()` call is made. The underlying messages are kept but they can be changed with the `update_MessageOfMemberOfGroup` call. The `SEND_ONCE` or `RECEIVE_ONCE` request are only handled once. Messages with the `SEND_ONCE` request are deleted after they are sent over the socket and the request of messages `RECEIVE_ONCE` is set to `NONE`. All messages with the `NONE` request are deleted at the beginning of the `communicate()` call. To retrieve a message from the library the `getMemberMessageOfGroup()` call can be used. `heliKomm` also provides functions to convert the message back to the original data type.

4. DISCUSSION

The aim of the `heliKomm` library is to improve the communication performance compared to a file based approach and also simplify the communication by providing simple API calls. To validate the efficiency two tests were performed: a synthetic test case and a strong coupling scenario with a NACA-0012 wing. For both tests local and remote performance were investigated.

The synthetic test case is designed to show the hypothetical performance of `heliKomm`. For that matter a fixed amount of data, i.e. 100 MB is sent and respectively received in differently sized data packets with no computation in between sending and receiving the data. The `TCP` and `FILE` protocols are benchmarked, both in their performance on a local machine as well as their remote operation. The `UNIX` protocol can only be used locally and is therefore only tested in that capacity. The `TCP` protocol is used locally by connecting to the ”localhost” domain. To test the remote performance of the `FILE` protocol the exchange directory is mounted with the `sshfs` command using the `cache=no` command line option to improve the performance. A maximal time limit of 120 s is imposed on all benchmarking runs. This time was only exceeded by the `FILE` protocol for small package sizes. The results of this synthetic test case are depicted in Figure 2 and Figure 3. Both bandwidth and transfer only take the actual communication into account not the creation and subsequent decoding of the message.

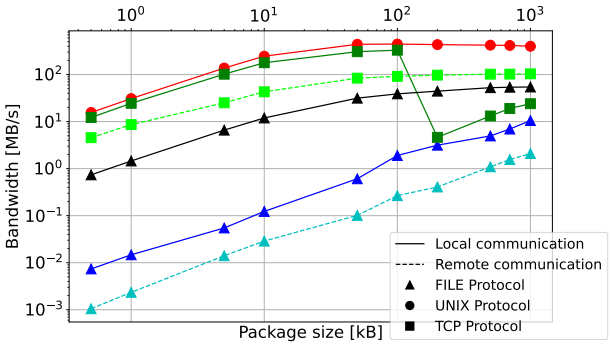

Figure 2. Comparison of the bandwidth of different socket protocols for a synthetic test case. Solid lines show the local communication, dashed lines are the remote connection. The black line shows the `FILE` protocol with a reduced file check time of 1 ms.

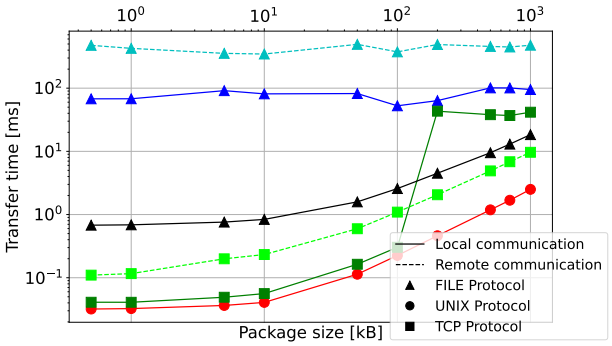

Figure 3. Comparison of the transfer times of different socket protocols for a synthetic test case. Solid lines show the local communication, dashed lines are the remote connection. The black line shows the `FILE` protocol with a reduced file check time of 1 ms.

From the results in Figure 2 it is evident that the `UNIX` protocol achieves the highest transfer bandwidth for all package sizes. It reaches values of up to 500 MB/s for package sizes of around 50 kB and keeps a similar performance up to package sizes of 1000 kB. For smaller packages, as also exhibited by the other protocols, the bandwidth decreases due to the increased frequency to which data is written to the socket and thus higher overhead in serializing the messages. The `TCP` protocol shows a similar behavior with a slightly decreased bandwidth for packages up to 100 kB. This can be attributed to the more complicated protocol underlying the `TCP` protocol. For larger packet sizes the bandwidth significantly drops, however, only for the local `TCP` connection. The reason for this behavior is not known but one possibility could be that the internal buffer which both client and server are using simultaneously is not large enough for packages larger than 100 kB. The `FILE` protocol exhibits the worst performance for all package sizes but especially for the smaller packages, which is nearly three orders or magnitude worse than for the `UNIX` or `TCP` protocol. The remote performance of the `TCP` and the `FILE` protocol show a similar trend as the local performance, albeit with a lower bandwidth. The significant drop observed in the local communication of the `TCP` protocol does, however, not occur for the remote communication. This enforces the assumption that an internal buffer is responsible for this decrease. The bandwidth for the remote `TCP` protocol plateaus at around 100 MB/s which is in line with the maximum network bandwidth available between the two machines. The bandwidth of the `FILE` protocol shows a nearly linear increase. This is due to the enforced wait time of 100 ms to check if a file is available, which dominates the overall execution time. This can be seen in the transfer times depicted in Figure 3 which remain constant around this transfer time. The black lines show the bandwidth and transfer time of the `FILE` protocol with a reduced wait time of 1 ms. The trend of the bandwidth is now comparable to the `UNIX` protocol, however, the performance stays worst for all package sizes. For an actual coupling scenario the bandwidth is only a secondary concern. As the amount of data exchanged is usually small the transfer time is more important. The transfer time is directly related to the bandwidth, so a high bandwidth means consequently a low transfer time. For all protocols, the transfer time is significantly less than the time necessary for a CFD time step, which means the overall runtime of the simulation is not significantly affected by the communication.

The NACA-0012 test case is designed to show the performance of `heliKomm` for an actual coupling scenario. The physical effects of this interaction are not investigated here. For that matter the wing structure is modeled as an Euler-Bernoulli beam where the material and geometric parameters are chosen so that visible deformations occur but still stiff enough to not produce unrealistic deformations. For the simulation the CFD solver FLOWer is coupled to the multi-body-dynamics solver SIMPACK. FLOWer calculates the aerodynamic forces, while SIMPACK calculates the deformation of the wing based on these loads. FLOWer uses the translation and rotation of the wing cross sections at different spanwise positions to deform first the CFD surface, and then a radial basis function based approach to extrapolate the deformations to the volume mesh, see Ref. 16. To show that the communication works as expected and the data is correctly exchanged, the deformed wing surfaces of SIMPACK and FLOWer are visually compared, c.f. Figure 4. The results show that the deformed CFD surface perfectly matches the discrete surface nodes of the SIMPACK solution. This shows that the approach used to deform the CFD mesh is consistent with the deformation calculated by SIMPACK and that the deformation data is transferred correctly.

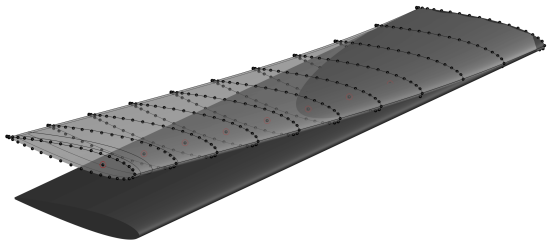

Figure 4. Closer look at the coupling results: dark grey: Undeformed CFD surface, light grey: deformed CFD surface, black dots: equivalent surface from SIMPACK, red circles: quarter chord positions from CFD

The message size for deformation as well as loads is directly proportional to the number of sections of the wing. For this setup 11 positions were chosen which results in messages of 602 bytes. The mean transfer times for the different protocols are depicted in Table 1. For comparison the wall time for one FLOWer time step is around 20 s and for the equivalent SIMPACK step around 1 ms. Comparing the transfer times of an actual coupling setup to the synthetic transfer times depicted in Figure 3, similar values are achieved for the `TCP` and `UNIX` protocol. The `FILE` protocol is faster than is expected from the results of the synthetic test case. This is due to the fact that SIMPACK provides the deformations before FLOWer reaches the point in the code where the data is necessary. Therefore, the enforced file check time of 100 ms is skipped more often. Reducing the wait time to 1 ms helps to reduce the transfer time but the performance gain is insignificant considering the overall simulation time.

Table 1. Mean transfer times for the different protocols achieved for the strong coupling test case.

Protocol	local	remote
`FILE`	2.2519 ms	15.1647 ms
`FILE 1ms`	0.1287 ms	
`TCP`	0.0577 ms	0.0334 ms
`UNIX`	0.0268 ms	

For a coupled simulation the choice of the communication protocol evidently does not significantly affect the overall simulation time of the simulation.

5. CONCLUSIONS

A lightweight socket based coupling library was developed for the task of efficiently exchanging data between two programs for the purpose of fluid structure coupling. A special Message protocol was developed to transmit the data between the programs. Furthermore, a group logic was developed to automatically handle the MPI communication with the master rank which also handles the socket communication.

As expected the socket based approach significantly increases the bandwidth of the communication. However, the effect for a realistic coupling scenario is not as pronounced. The exchange times of all protocols are significantly less than the time necessary for a CFD time step. Therefore, a file based data exchange, although not as fast, is a viable communication strategy, as long as the data is handled in a binary format. For the remote communication a mounted file systems also provides a viable alternative to a more complex direct file transfer. Here, however, the performance is highly influenced by the underlying file system.

The main benefit of `heliKomm` compared to a specifically designed approach is the ease of use. The complete coupling exchange can be done with a few API calls.

Author contact:
Julius Klauck, julius.klauck@iag.uni-stuttgart.de
Manuel Keßler, kessler@iag.uni-stuttgart.de

6. APPENDIX

Table 2. API Reference: All function calls are member calls of the CommunicationManager object.

Function	Description	Parameters	Called by
`createFromInputFile`	start a server or client with a specific protocol from a configuration file	filename	MPI Master rank
`newGroup`	create a new communication group	group name	every MPI rank
`addMemberToGroup`	add a communication member to the group	group name; member name	Any rank who needs to communicate in this group
`removeMemberFromGroup`	remove a communication member from the group	group name; member name	any rank who has added a member to this group
`finalizeGroup`	finish the group logic and find all ranks which are part of this group	group name	every MPI rank
`add_MessageToMemberOfGroup`	add a message to a member of a group	group name; member name; message; communication request	any rank who has added this member to the group
`remove_MessageFromMemberOfGroup`	add a message to a member of a group	group name; member name; message; communication request	any rank who has added this member to the group
`communicate`	start the complete communication phase	group name	every rank that is part of the group
`getMemberMessageOfGroup`	get the message of a member of a group	group name; member name; message header	any rank that is part of the group

7. REFERENCES

1. K. Pahlke and B. Wall, “Calculation of multi-bladed rotors in high-speed forward flight with weak fluid-structure-coupling,” in *27th European Rotorcraft Forum, Moscow, Russia*, pp. 27.1–27.11.

2. W. Johnson, *CAMRAD II Comprehensive analytical model of rotorcraft aerodynamics and dynamics*, 4th ed.

3. B. Benoit, A.-M. Dequin, K. Kampa, W. von Grünhagen, P.-M. Basset, and B. Gimonet, “HOST, a general helicopter simulation tool for Germany and France.”

4. J. Letzgus, M. Keßler, and E. Krämer, “Simulation of dynamic stall on an elastic rotor in high-speed turn flight,” in *75th VFS Annual Forum, Philadelphia, PA*.

5. F. Frey, C. Öhrle, J. Thiemeier, M. Keßler, and E. Krämer, “Aerodynamic interactions on Airbus Helicopters’ compound helicopter RACER in hover,” in *76th VFS Annual Forum, Virginia Beach, VA (virtual event)*.

6. M. Embacher, M. Keßler, M. Dietz, and E. Krämer, “Coupled CFD-simulation of a helicopter in free-flight trim,” vol. 1, pp. 730–744.

7. A. D’Alascio, S. Link, T. Ries, T. Kneisch, and D. Schimke, “New role of CFD in the helicopter design process - the EC145 T2 experience,” in *39th European Rotorcraft Forum, Moscow, Russia*.

8. P. Kunze and T. Ries, “Quasi-steady aeromechanic helicopter simulations using mid-fidelity aerodynamics,” in *78th VFS Annual Forum, Fort Worth, TX*.

9. A. Altmikus, S. Wagner, P. Beaumier, and G. Servera, “A comparison: Weak versus strong modular coupling for trimmed aeroelastic rotor simulations.”

10. P. Beaumier, M. Costes, B. Rodriguez, M. Poinot, and B. Cantaloube, “Weak and strong coupling between the elsA CFD solver and the HOST helicopter comprehensive analysis,” in *31st European Rotorcraft Forum, Florence, Italy*.

11. B. Gatzhammer, M. Mehl, and T. Neckel, “A coupling environment for partitioned multiphysics simulations applied to fluid-structure interaction scenarios,” vol. 1, no. 1, pp. 681–689, iCCS 2010.

12. GNU. The GNU C library (glibc). [Online]. Available: https://www.gnu.org/savannah-checkouts/gnu/libc/manual/html_node/Sockets.html [Accessed: 01 August 2024].

13. man7. Linux man pages online. [Online]. Available: https://man7.org/linux/man-pages/man7/unix.7.html [Accessed: 01 August 2024].

14. ——. Linux man pages online. [Online]. Available: https://man7.org/linux/man-pages/man7/tcp.7.html [Accessed: 01 August 2024].

15. MPI. MPI forum. [Online]. Available: https://www.mpi-forum.org/ [Accessed: 01 August 2024].

16. M. Schuff, P. Kranzinger, M. Keßler, and E. Krämer, “Advanced CFD-CSD coupling: Generalized, high performant, radial basis function based volume mesh deformation algorithm for structured, unstructured, and overlapping meshes,” in *40th European Rotorcraft Forum, Southampton, UK*, 2014.