RUST bindings for DIM

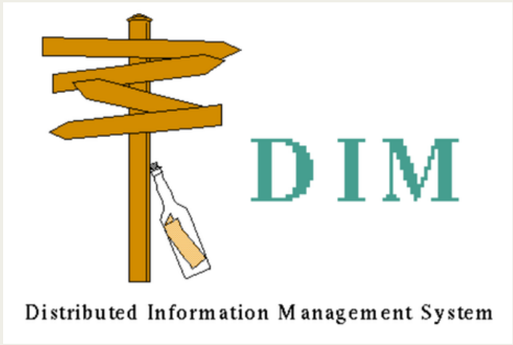
BETÜL DOĞRUL
SUPERVISORS: NIKO NEUFELD, TOMMASO COLOMBO, FLAVIO PISANI

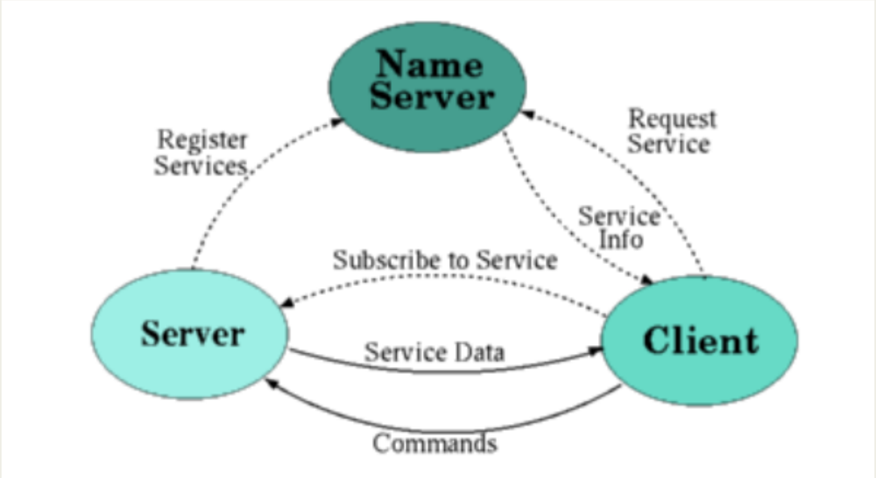
AGENDA

- What is DIM?
- Why RUST?
- The Challenge
- My Approach
- Key Features Implemented
- Benefits of RUST with DIM
- Challenges Faced
- Future Work
- Conclusion

WHAT IS DIM?

- A communication system for distributed/mixed environments
- Network transparent inter-process communication layer.
- Client/server paradigm
- Servers - services and publish information.
- Clients subscribe to these services or send commands.
- Widely used at CERN for monitoring, control systems, and data acquisition systems etc.





WHY RUST?

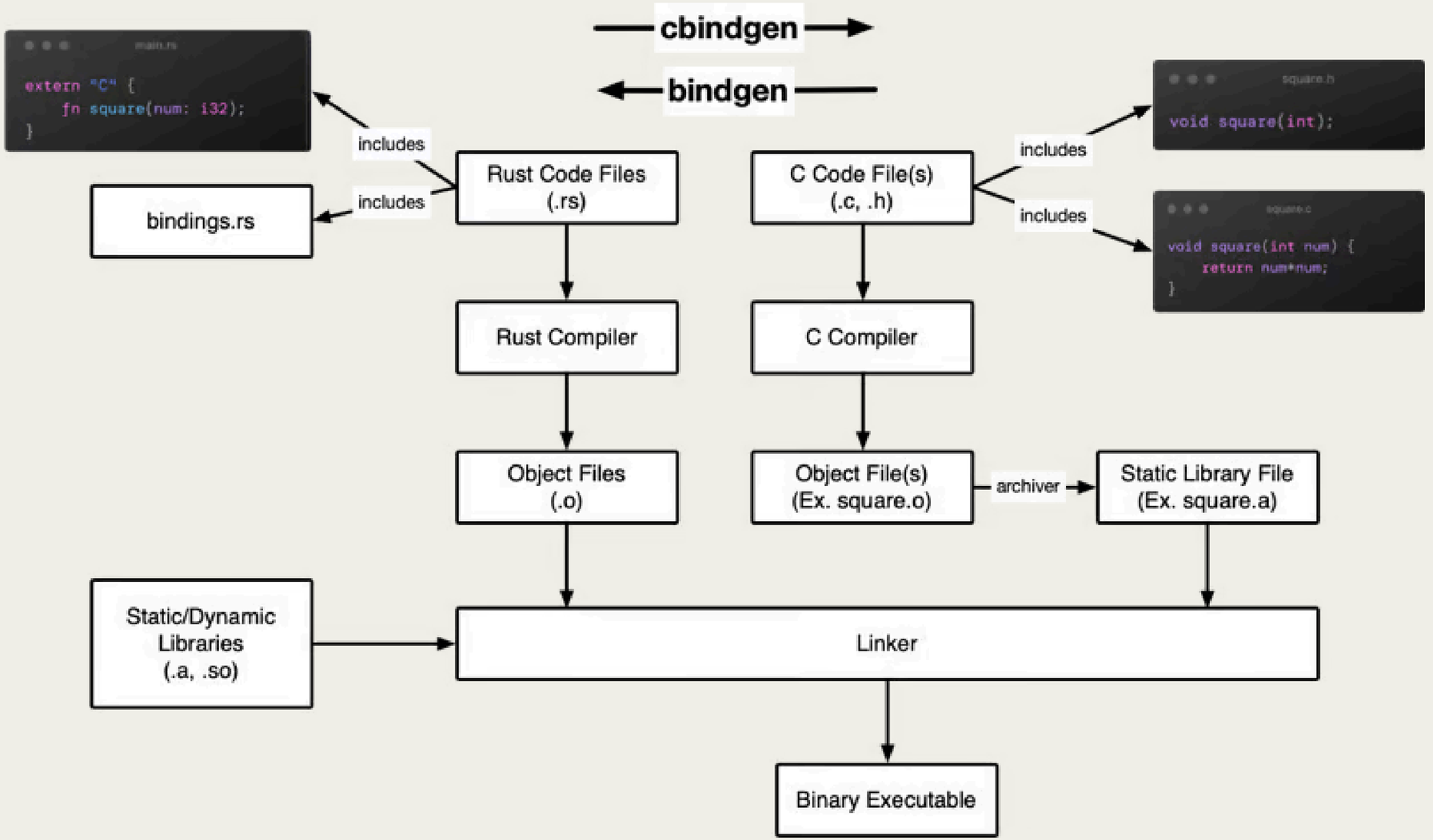
- A modern systems programming language
- **Memory Safety** - prevents common memory-related bugs such as buffer overflows and use-after-free errors without needing a garbage collector.
- **Concurrency Safety** - ownership model ensures thread safety, preventing data races and ensuring reliable concurrent operations.
- **Performance** - offers low-level control over memory while maintaining high performance, comparable to C, which is crucial for high-demand systems like DIM.

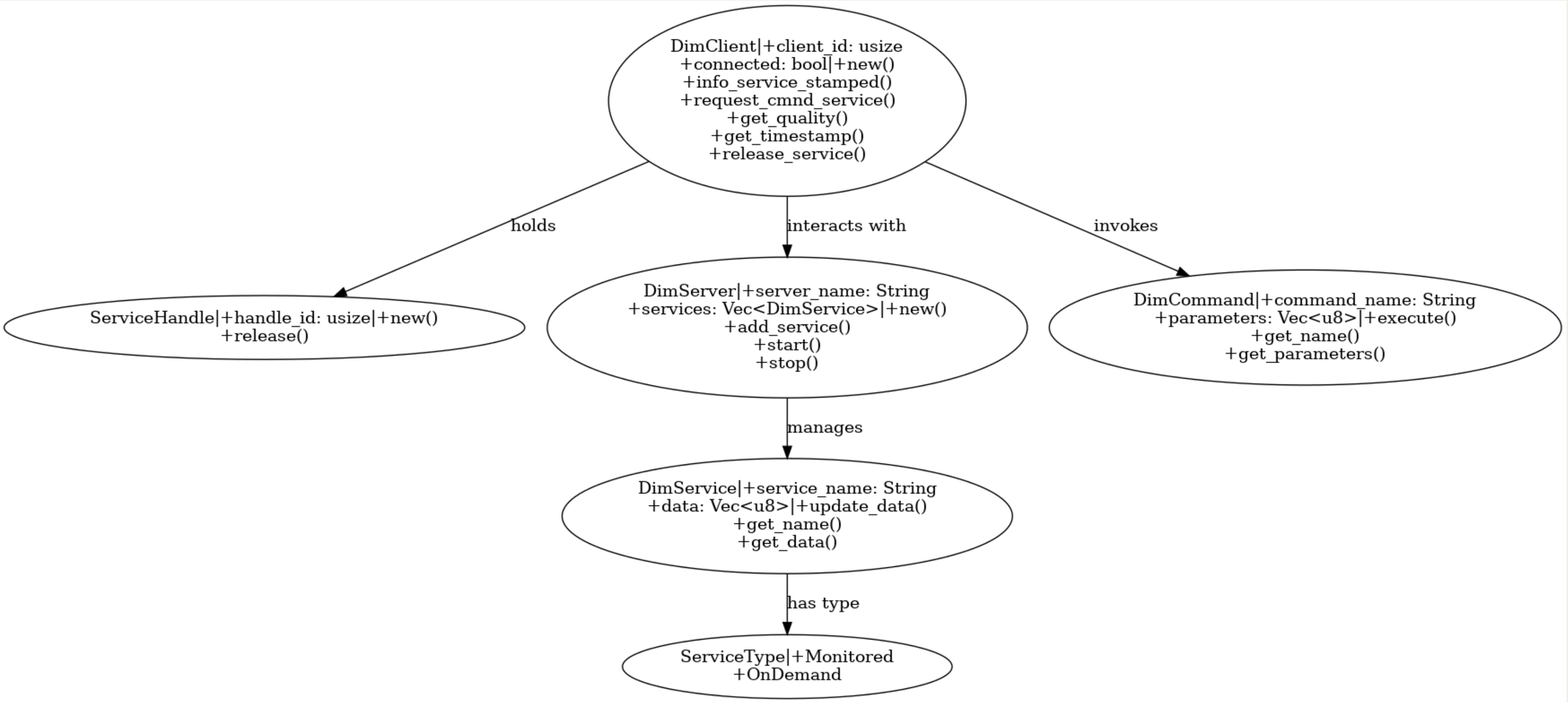
THE CHALLENGE

- Different Memory Management Models
  - DIM is written in C -> direct access to memory and relies heavily on manual management
  - Rust -> strict rules around memory access and safety.
- Callback Mechanisms:
  - DIM uses callbacks to notify clients of events.
  - Rust’s closure and function pointer systems don’t map directly to C’s callback mechanisms, requiring a custom solution to manage lifetimes and safety.
- Multithreading
  - DIM clients operate concurrently, managing multiple services at once.
  - Rust’s concurrency model -> Arc and Mutex, needed to be integrated with DIM’s threading model.

MY APPROACH

- Understanding **DIM API**
- **Using bindgen -** generates Rust FFI (Foreign Function Interface) bindings to C libraries
- Manual Refinement
- **Manage Pointers -** Encapsulate unsafe C pointers in Rust-safe abstractions to prevent memory issues.
- **Handle Callbacks -** Develop a system to safely pass Rust closures to C, ensuring proper lifetime management
- **Ensure Concurrency Safety -** Adapt DIM’s concurrency features to fit Rust’s ownership and concurrency models, ensuring thread safety.


cbindgen
bindgen
main.rs
extern "C" {
fn square(num: i32);
}
includes
includes
bindings.rs
Rust Code Files
(.rs)
C Code File(s)
(.c, .h)
includes
includes
square.h
void square(int);
square.c
void square(int num) {
return num*num;
}
Rust Compiler
C Compiler
Object Files
(.o)
Object File(s)
(Ex. square.o)
archiver
Static Library File
(Ex. square.a)
Static/Dynamic
Libraries
(.a, .so)
Linker
Binary Executable


DimClient|+client_id: usize
+connected: bool|+new()
+info_service_stamped()
+request_cmnd_service()
+get_quality()
+get_timestamp()
+release_service()
holds
interacts with
invokes
ServiceHandle|+handle_id: usize|+new()
+release()
DimServer|+server_name: String
+services: Vec<DimService>|+new()
+add_service()
+start()
+stop()
DimCommand|+command_name: String
+parameters: Vec<u8>|+execute()
+get_name()
+get_parameters()
manages
DimService|+service_name: String
+data: Vec<u8>|+update_data()
+get_name()
+get_data()
has type
ServiceType|+Monitored
+OnDemand

KEY FEATURES IMPLEMENTED

```rust
impl DimServer {
    pub fn new() -> Self {
        DimServer { task_name: None }
    }

    pub fn add_service(
        &self,
        service_name: &str,
        service_type: DimFormat,
        service_address: *mut c_void,
        service_size: i32,
        usr_routine: Option<
            unsafe extern "C" fn(*mut c_void, *mut *mut c_void, *mut i32, *mut i32),
        >,
        tag: i64,
    ) -> Result<DimService, &'static str> {
        // Debug messages
        println!("Creating new DimService");
        println!("  Service Name: {}", service_name);
        println!("  Service Type: {:?}", service_type);
        println!("  Service Address: {:?}", service_address);
        println!("  Service Size: {}", service_size);
        println!("  Tag: {}", tag);

        let c_name: CString = CString::new(service_name).expect(msg: "Invalid service name");
        println!("  C String Name: {:?}", c_name);
        let str_description: String = service_type.to_string();
        let c_type: CString = CString::new(str_description).expect(msg: "Invalid service type");
        println!("  C String Type: {:?}", c_type);

        let service_id: u32 = unsafe {
            dis_add_service_(
                service_name: c_name.as_ptr(),
                service_type: c_type.as_ptr(),
                service_address,
                service_size,
                usr_routine,
                tag,
            )
        };

        println!("  Service ID: {}", service_id);

        if service_id != 0 {
            println!("  DimService created successfully with ID: {}", service_id);
            Ok(DimService {
                service_id: Some(service_id),
                service_name: c_name,
                service_address,
                service_type,
                usr_routine,
                tag,
            })
        } else {
            println!("  Failed to create DimService");
            Err("Failed to create DimService")
        }
    } fn add_service
```

- **Service Subscription -** allows Rust clients to subscribe to services provided by DIM servers, a safe API

KEY FEATURES IMPLEMENTED

```rust
pub fn request_cmnd_service(
    &self,
    command_name: &str,
    serv_address: &mut [u8],
) -> Result<(), CommandError> {
    // Convert command_name to a C-compatible string
    let c_command_name: CString =
        CString::new(command_name).map_err(op: |_| CommandError::ConversionError)?;

    let serv_size: i32 = serv_address.len() as i32;

    // Call the external C function
    let ret: i32 = unsafe {
        dic_cmnd_service_(
            service_name: c_command_name.as_ptr(),
            service_address: serv_address.as_mut_ptr() as *mut c_void,
            service_size: serv_size,
        )
    };

    if ret == 1 {
        Ok(())
    } else {
        Err(CommandError::CommandNotFound)
    }
}
```

- **Command Handling** - Clients can request execution of commands to DIM servers using high-level Rust constructs

KEY FEATURES IMPLEMENTED

- **Thread-Safe Clients** - Enabled concurrent operations using Arc and Mutex, allowing multiple threads to interact with DIM simultaneously.
- **Error Handling Enhancements** - Rust's Result type - provides more descriptive and manageable error reporting.
- **Resource Management** - Rust's Drop trait for automatic resource cleanup, preventing leaks and ensuring efficient resource management.

```rust
pub fn start_serving(&mut self, task_name: &str) -> Result<i32, ServingError> {
    if self.task_name == None {
        println!("Starting serving with task_name {}", task_name);
        let c_task_name: CString =
            CString::new(task_name).map_err(op: |_| ServingError::TaskNameConversionFailed)?;
        println!("  C String Task Name: {:?}", c_task_name);

        self.task_name = Some(c_task_name.clone());

        let result: i32 = unsafe { dis_start_serving_(task_name: c_task_name.as_ptr()) };
        println!("  Start serving result: {}", result);
        match result {
            1 => Ok(1),
            0 => Err(ServingError::InvalidServiceId),
            x: i32 => Ok(x),
        }
    } else {
        println!("Server has already started serving!\n");
        Err(ServingError::AlreadyServingError)
    }
}
```

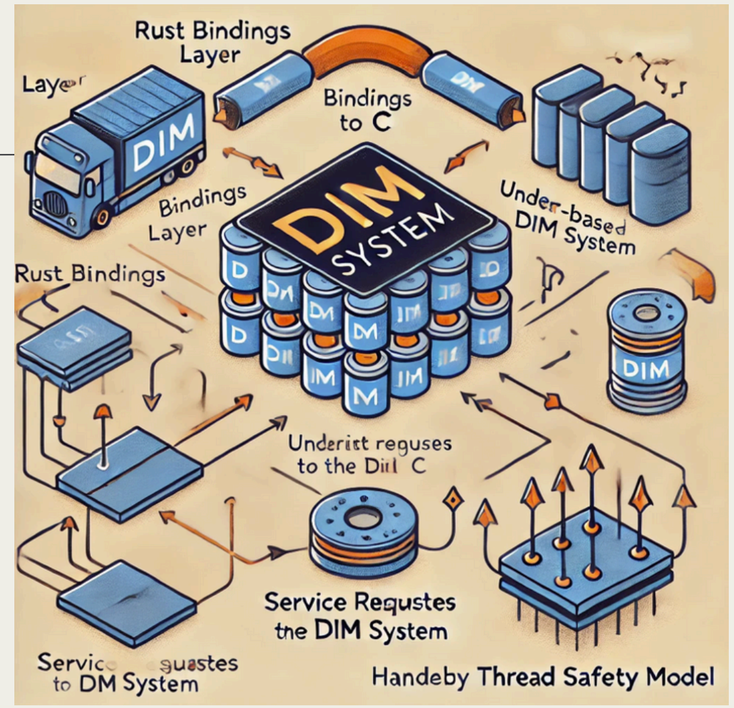
```rust
impl Drop for DimServer {
    fn drop(&mut self) {
        if self.task_name.is_some() {
            println!("Dropping DimServer: stopping serving.");
            self.stop_serving();
        } else {
            println!("Dropping DimServer: no active service to stop.");
        }
    }
}
```

BENEFITS OF RUST WITH DIM

- **Memory Safety**: By preventing common memory issues, the bindings improve the reliability of DIM operations.
- **Concurrency Safety**: Rust's model helps avoid data races and concurrency issues, ensuring safer multithreaded operations.
- **Performance**: Rust maintains high performance, similar to C, while providing additional safety features.
- **Ease of Use**: The bindings provide an idiomatic Rust API, simplifying interaction with DIM and making it more accessible for Rust developers.

CHALLENGES FACED

- **FFI Layer Complexity**: Handling the foreign function interface (FFI) required careful management of raw pointers and memory.
- **Debugging Issues**: Debugging issues related to unsafe code involved both Rust and C debugging tools, which was complex and time-consuming.
- **Thread Safety**: Ensuring compatibility between DIM's threading model and Rust's concurrency model required careful design and implementation.



FUTURE WORK

- **API Refinements**: Some APIs could be made more idiomatic and user-friendly.
- **Callback** function structure improvements
- **Performance Optimizations**: Further optimizations could enhance the performance of the bindings.

CONCLUSION

- Modernizing and enhancing DIM by integrating Rust’s safety and concurrency features
- The Rust bindings provide a safer, more efficient interface for interacting with DIM
- Q&A

Thank you!