Tic-Tac-Toe

This guide is rated as basic .

You can expect basic guides to take 30-45 minutes of dedicated time. The length of time necessary to fully understand some of the concepts raised in this guide might increase this estimate.

You can view the complete source code for this app example in the Sui repository.

This guide covers three different implementations of the game tic-tac-toe on Sui. The first example utilizes a centralized admin that owns the board object and marks it on the users' behalf. The second example utilizes a shared object that both users can mutate. And the third example utilizes a multisig , where instead of sharing the game board, it's in a 1-of-2 multisig of both users' accounts.

The guide is divided into three parts that each cover a different implementation of the tic-tac-toe game board:

Before getting started, make sure you have:

To begin, create a new folder on your system titled tic-tac-toe that holds all your files.

In this folder, create the following subfolders:

Add Move.toml to tic-tac-toe/move/

Create a new file in tic-tac-toe/move/sources titled owned.move . Later, you will update this file to contain the Move code for the game board in the centralized (and multisig) version of tic-tac-toe.

In this first example of tic-tac-toe, the Game object, including the game board, is controlled by a game admin.

Ignore the admin field for now, as it is only relevant for the multisig approach.

Games are created with the new function:

Some things to note:

Because the players don't own the game board object, they cannot directly mutate it. Instead, they indicate their move by creating a Mark object with their intended placement and send it to the game object using transfer to object:

When playing the game, the admin operates a service that keeps track of marks using events. When a request is received (send_mark), the admin tries to place the marker on the board (place_mark). Each move requires two steps (thus two transactions): one from the player and one from the admin. This setup relies on the admin's service to keep the game moving.

When a player sends a mark, a Mark object is created and is sent to the Game object. The admin then receives the mark and places it on the board. This is a use of dynamic object fields, where an object, Game , can hold other objects, Mark .

To view the entire source code, see the owned.move source file . You can find the rest of the logic, including how to check for a winner, as well as deleting the game board after the game concludes there.

owned.move

An alternative version of this game, shared tic-tac-toe, uses shared objects for a more straightforward implementation that doesn't use a centralized service. This comes at a slightly increased cost, as using shared objects is more expensive than transactions involving wholly owned objects.

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

What the guide teaches

Before getting started, make sure you have:

To begin, create a new folder on your system titled tic-tac-toe that holds all your files.

In this folder, create the following subfolders:

Add Move.toml to tic-tac-toe/move/

Create a new file in tic-tac-toe/move/sources titled owned.move . Later, you will update this file to contain the Move code for the game board in the centralized (and multisig) version of tic-tac-toe.

In this first example of tic-tac-toe, the Game object, including the game board, is controlled by a game admin.

Ignore the admin field for now, as it is only relevant for the multisig approach.

Games are created with the new function:

Some things to note:

Because the players don't own the game board object, they cannot directly mutate it. Instead, they indicate their move by creating a Mark object with their intended placement and send it to the game object using transfer to object:

When playing the game, the admin operates a service that keeps track of marks using events. When a request is received (send_mark), the admin tries to place the marker on the board (place_mark). Each move requires two steps (thus two transactions): one from the player and one from the admin. This setup relies on the admin's service to keep the game moving.

When a player sends a mark, a Mark object is created and is sent to the Game object. The admin then receives the mark and places it on the board. This is a use of dynamic object fields, where an object, Game , can hold other objects, Mark .

To view the entire source code, see the owned.move source file . You can find the rest of the logic, including how to check for a winner, as well as deleting the game board after the game concludes there.

owned.move

An alternative version of this game, shared tic-tac-toe, uses shared objects for a more straightforward implementation that doesn't use a centralized service. This comes at a slightly increased cost, as using shared objects is more expensive than transactions involving wholly owned objects.

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

What you need

Before getting started, make sure you have:

To begin, create a new folder on your system titled tic-tac-toe that holds all your files.

In this folder, create the following subfolders:

Add Move.toml to tic-tac-toe/move/

Create a new file in tic-tac-toe/move/sources titled owned.move . Later, you will update this file to contain the Move code for the game board in the centralized (and multisig) version of tic-tac-toe.

In this first example of tic-tac-toe, the Game object, including the game board, is controlled by a game admin.

Ignore the admin field for now, as it is only relevant for the multisig approach.

Games are created with the new function:

Some things to note:

Because the players don’t own the game board object, they cannot directly mutate it. Instead, they indicate their move by creating a Mark object with their intended placement and send it to the game object using transfer to object:

When playing the game, the admin operates a service that keeps track of marks using events. When a request is received (send_mark), the admin tries to place the marker on the board (place_mark). Each move requires two steps (thus two transactions): one from the player and one from the admin. This setup relies on the admin's service to keep the game moving.

When a player sends a mark, a Mark object is created and is sent to the Game object. The admin then receives the mark and places it on the board. This is a use of dynamic object fields, where an object, Game , can hold other objects, Mark .

To view the entire source code, see the [owned.move source file](owned.move) . You can find the rest of the logic, including how to check for a winner, as well as deleting the game board after the game concludes there.

owned.move

An alternative version of this game, shared tic-tac-toe, uses shared objects for a more straightforward implementation that doesn't use a centralized service. This comes at a slightly increased cost, as using shared objects is more expensive than transactions involving wholly owned objects.

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

Directory structure

To begin, create a new folder on your system titled tic-tac-toe that holds all your files.

In this folder, create the following subfolders:

Add Move.toml to tic-tac-toe/move/

Create a new file in tic-tac-toe/move/sources titled owned.move . Later, you will update this file to contain the Move code for the game board in the centralized (and multisig) version of tic-tac-toe.

In this first example of tic-tac-toe, the Game object, including the game board, is controlled by a game admin.

Ignore the admin field for now, as it is only relevant for the multisig approach.

Games are created with the new function:

Some things to note:

Because the players don't own the game board object, they cannot directly mutate it. Instead, they indicate their move by creating a Mark object with their intended placement and send it to the game object using transfer to object:

When playing the game, the admin operates a service that keeps track of marks using events. When a request is received (send_mark), the admin tries to place the marker on the board (place_mark). Each move requires two steps (thus two transactions): one from the player and one from the admin. This setup relies on the admin's service to keep the game moving.

When a player sends a mark, a Mark object is created and is sent to the Game object. The admin then receives the mark and places it on the board. This is a use of dynamic object fields, where an object, Game , can hold other objects, Mark .

To view the entire source code, see the owned.move source file . You can find the rest of the logic, including how to check for a winner, as well as deleting the game board after the game concludes there.

owned.move

An alternative version of this game, shared tic-tac-toe, uses shared objects for a more straightforward implementation that doesn't use a centralized service. This comes at a slightly increased cost, as using shared objects is more expensive than transactions involving wholly owned objects.

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

owned.move

Create a new file in tic-tac-toe/move/sources titled owned.move . Later, you will update this file to contain the Move code for the game board in the centralized (and multisig) version of tic-tac-toe.

In this first example of tic-tac-toe, the Game object, including the game board, is controlled by a game admin.

Ignore the admin field for now, as it is only relevant for the multisig approach.

Games are created with the new function:

Some things to note:

Because the players don't own the game board object, they cannot directly mutate it. Instead, they indicate their move by creating a Mark object with their intended placement and send it to the game object using transfer to object:

When playing the game, the admin operates a service that keeps track of marks using events. When a request is received (send_mark), the admin tries to place the marker on the board (place_mark). Each move requires two steps (thus two transactions): one from the player and one from the admin. This setup relies on the admin's service to keep the game moving.

When a player sends a mark, a Mark object is created and is sent to the Game object. The admin then receives the mark and places it on the board. This is a use of dynamic object fields, where an object, Game , can hold other objects, Mark .

To view the entire source code, see the owned.move source file . You can find the rest of the logic, including how to check for a winner, as well as deleting the game board after the game concludes there.

owned.move

An alternative version of this game, shared tic-tac-toe, uses shared objects for a more straightforward implementation that doesn't use a centralized service. This comes at a slightly increased cost, as using shared objects is more expensive than transactions involving wholly owned objects.

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

shared.move

In the previous version, the admin owned the game object, preventing players from directly changing the gameboard, as well as requiring two transactions for each marker placement. In this version, the game object is a shared object, allowing both players to access and modify it directly, enabling them to place markers in just one transaction. However, using a shared object generally incurs extra costs because Sui needs to sequence the operations from different transactions. In the context of this game, where players are expected to take turns, this shouldn't significantly impact performance. Overall, this shared object approach simplifies the implementation compared to the previous method.

As the following code demonstrates, the Game object in this example is almost identical to the one before it. The only differences are that it does not include an admin field, which is only relevant for the multisig version of the game, and it does not have store , because it only ever exists as a shared object (so it cannot be transferred or wrapped).

Take a look at the new function:

Instead of the game being sent to the game admin, it is instantiated as a shared object. The other notable difference is that there is no need to mint a TurnCap because the only two addresses that can play this game are x and o , and this is checked in the next function, place_mark :

shared.move

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .

Multisig

Multisig tic-tac-toe uses the same Move code as the owned version of the game, but interacts with it differently. Instead of transferring the game to a third party admin account, the players create a 1-of-2 multisig account to act as the game admin, so that either player can sign on behalf of the "admin". This pattern offers a way to share a resource between up to ten accounts without relying on consensus.

In this implementation of the game, the game is in a 1-of-2 multisig account that acts as the game admin. In this particular case, because there are only two players, the previous example is a more convenient use case. However, this example illustrates that in some cases, a multisig can replace shared objects, thus allowing transactions to bypass consensus when using such an implementation.

A multisig account is defined by the public keys of its constituent keypairs, their relative weights, and the threshold -- a signature is valid if the sum of weights of constituent keys having signed the signature exceeds the threshold. In our case, there are at most two constituent keypairs, they each have a weight of 1 and the threshold is also 1. A multisig cannot mention the same public key twice, so keys are deduplicated before the multisig is formed to deal with the case where a player is playing themselves:

MultiSig.ts

Note that an address on Sui can be derived from a public key (this fact is used in the previous example to deduplicate public keys based on their accompanying address), but the opposite is not true. This means that to start a game of multisig tic-tac-toe, players must exchange public keys, instead of addresses.

When creating a multisig game, we make use of owned::Game 's admin field to store the multisig public key for the admin account. Later, it will be used to form the signature for the second transaction in the move. This does not need to be stored on-chain, but we are doing so for convenience so that when we fetch the Game 's contents, we get the public key as well:

useTransactions.ts also contains functions to place, send, and receive marks, end the game, and burn completed games. These functions all return a Transaction object, which is used in the React frontend to execute the transaction with the appropriate signer.

useTransactions.ts

Placing a mark requires two transactions, just like the owned example, but they are both driven by one of the players. The first transaction is executed by the player as themselves, to send the mark to the game, and the second is executed by the player acting as the admin to place the mark they just sent. In the React frontend, this is performed as follows:

Game.tsx

The first step is to get the multisig public key, which was written to Game.admin earlier. Then two executor hooks are created: The first is to sign and execute as the current player, and the second is to sign and execute as the multisig/admin account. After the wallet has serialized and signed the transaction the second executor creates a multisig from the wallet signature and executes the transaction with two signatures: Authorizing on behalf of the multisig and the wallet.

The reason for the two signatures is clearer when looking at the construction of the recv transaction: The multisig authorizes access to the Game , and the wallet authorizes access to the gas object. This is because the multisig account does not hold any coins of its own, so it relies on the player account to sponsor the transaction.

You can find an example React front-end supporting both the multi-sig and shared variants of the game in the ui directory , and a CLI written in Rust in the cli directory .